# SpringCloud应该怎么入门

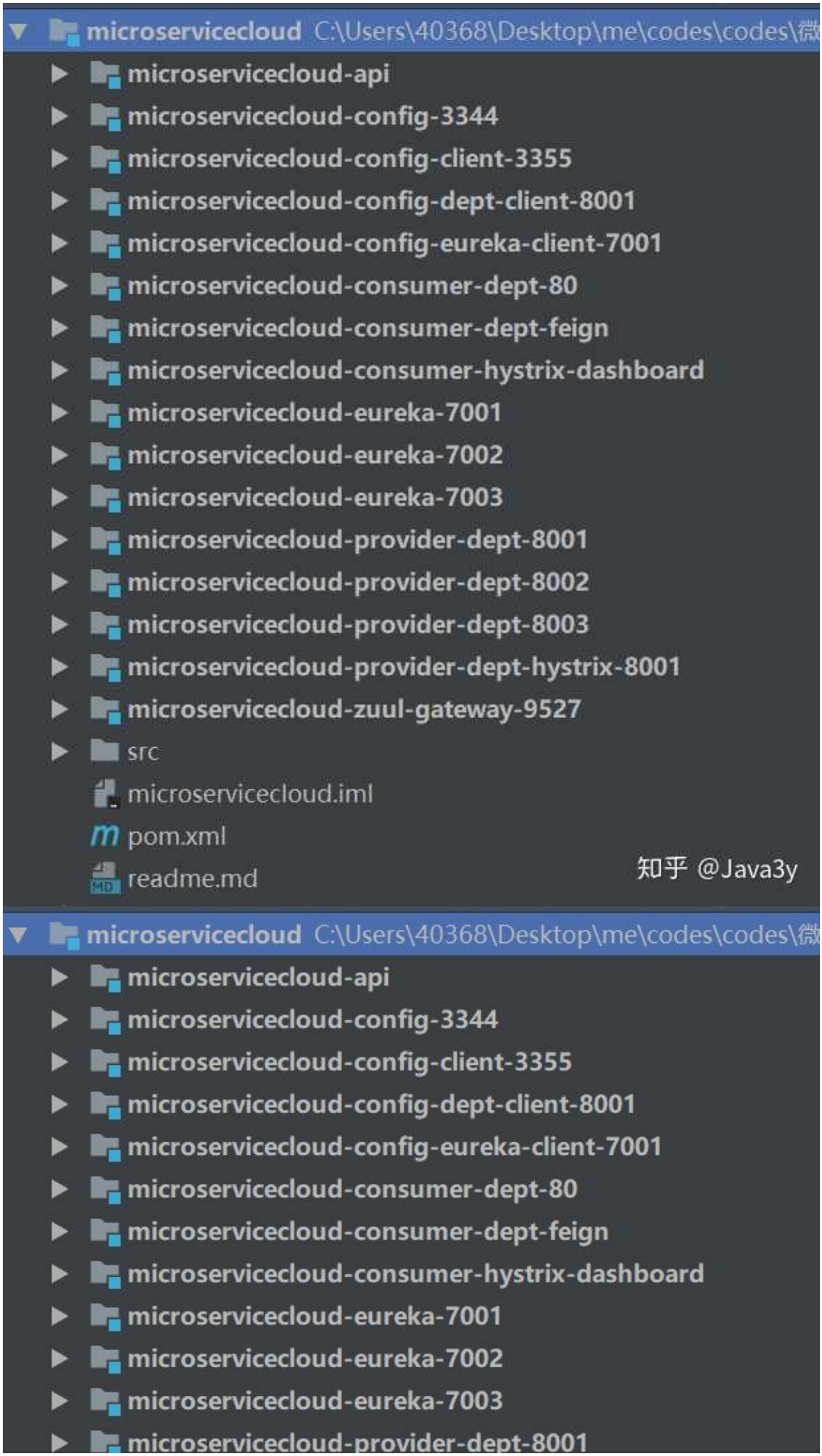
## 1 SpringCloud GitHub Demo

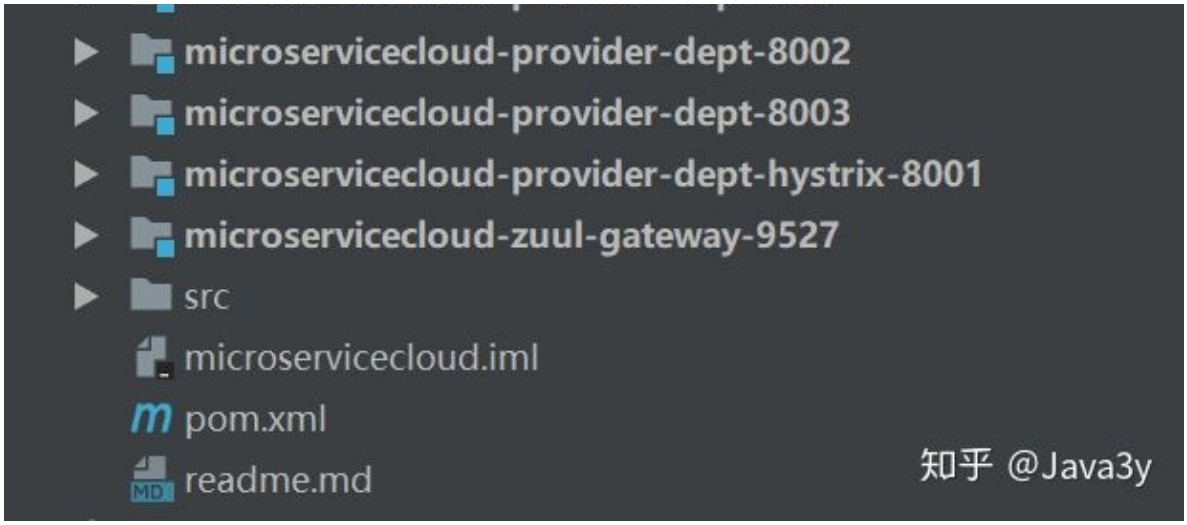
我这边在学习的时候写过一篇SpringCloud文章，题主可以看看（应该还算通俗易懂的）

SpringCloud GitHub Demo(看完文章的同学可以自己练手玩玩)：

- https://github.com/ZhongFuCheng3y/msc-Demo

项目结构图：


microservicecloud C:\Users\40368\Desktop\me\codes\codes\微
microservicecloud-api
microservicecloud-config-3344
microservicecloud-config-client-3355
microservicecloud-config-dept-client-8001
microservicecloud-config-eureka-client-7001
microservicecloud-consumer-dept-80
microservicecloud-consumer-dept-feign
microservicecloud-consumer-hystrix-dashboard
microservicecloud-eureka-7001
microservicecloud-eureka-7002
microservicecloud-eureka-7003
microservicecloud-provider-dept-8001
microservicecloud-provider-dept-8002
microservicecloud-provider-dept-8003
microservicecloud-provider-dept-hystrix-8001
microservicecloud-zuul-gateway-9527
src
microservicecloud.iml
pom.xml
readme.md
知乎 @Java3y
microservicecloud C:\Users\40368\Desktop\me\codes\codes\微
microservicecloud-api
microservicecloud-config-3344
microservicecloud-config-client-3355
microservicecloud-config-dept-client-8001
microservicecloud-config-eureka-client-7001
microservicecloud-consumer-dept-80
microservicecloud-consumer-dept-feign
microservicecloud-consumer-hystrix-dashboard
microservicecloud-eureka-7001
microservicecloud-eureka-7002
microservicecloud-eureka-7003
microservicecloud-provider-dept-8001




# 2 集群/分布式/微服务/SOA是什么?

像我这种技术小白，看到这些词(`集群/分布式/微服务/SOA`)的时候，感觉就是遥不可及的(高大尚的技术！！)。就好像刚学Java面向对象的时候，在论坛上翻阅资料的时候，无意看到"面向切面编程"，也认为这是遥不可及的(高大尚的技术！！)。

但真正接触到"面向切面编程"的时候，发现原来就是如此啊，也没什么大不了的。只不过当时被它的名字给唬住了...

不知道各位在刚接触这些名字`集群/分布式/微服务/SOA`的时候，有没有被唬住了呢？？

- 下面我就**简单**说说这些名词的意思

## 2.1什么是集群

以下内容来源维基百科：

> 计算机集群简称集群是一种计算机系统，它通过一组松散集成的计算机软件和/或硬件连接起来高度紧密地协作完成计算工作。在某种意义上，他们可以被看作是一台计算机。集群系统中的单个计算机通常称为节点，通常通过局域网连接，但也有其它的可能连接方式。集群计算机通常用来改进单个计算机的计算速度和/或可靠性。一般情况下集群计算机比单个计算机，比如工作站或超级计算机性能价格比要高得多

集群技术特点：

- 通过**多台计算机完成同一个工作**，达到更高的效率。
- **两机或多机内容、工作过程等完全一样。**如果一台死机，另一台可以起作用。

在维基百科上说得也挺明白的了，我来举个**例子**吧。

- 小周在公司写Java程序，但公司业务在发展，一个Java开发者可能**忙不过来**，小周有的时候也得**请个假呀。**于是请了3y过去**一起做Java开发。**平时小周和3y就写Java程序，但3y可能**有事**要回学校一趟。没事，公司还有小周做Java开发呢，公司开发还能继续运作。
    - 3y跟小周**都是做Java开发。**
    - 3y来了，小周的工作可以**分担**一些。
    - 3y请假了，还有小周在呢。

我写了一个910便利网发布到服务器去了，现在越来越多的人访问了，访问有点慢，怎么办？？？很简单，(只有充钱才能变强)，加配置吧(加cpu，加内存)。升级完配置之后，访问人数越来越多，于是发现又不禁用啦，在这台机器上加配置已经解决不了了，怎么办？？？很简单，(只有充钱才能变强)，我**再买一台服务器，将910便利网也发布到新买的这台服务器上去。**

特点：

- 这两台服务器都是运行**同一个系统**--->910便利网

好处：

- 本来只有一台机器处理访问，现在有两台机器处理访问了，**分担了压力。**
- 如果其中一台忘记缴费了，暂时用不了了。没关系，还有另一台可以用呢。

**集群**：同一个业务，部署在多个服务器上(不同的服务器运行同样的代码，干同一件事)

## 2.2什么是分布式

以下内容来源维基百科：

> 分布式系统是一组计算机，通过网络相互连接传递消息与通信后并协调它们的行为而形成的系统。**组件之间彼此进行交互以实现一个共同的目标。**

我也来举个例子来说明一下吧：

- 现在公司有小周和3y一起做Java开发，做Java开发一般jQuery，AJAX都能写一点，所以这些活都由我们来干。可是呢，3y对前端不是很熟，有的时候调试半天都调不出来。老板认为3y是真的菜！于是让小周**专门来处理前端**的事情。这样3y就高兴了，可以**专心写自己的Java**，前端就**专门**交由小周负责了。于是，小周和3y就变成了**协作开发。**
    - 3y对前端不熟(能写出来)，但在调试的时候可能会花费很多时间
    - 小周来**专门做前端**的事，3y可以**专心写自己的Java程序**了。
    - 都是为了项目正常运行以及迭代。

我的910便利网已经部署到两台服务器去了，但是越来越多的人去访问。现在也逐渐承受不住啦。那现在怎么办啊？？那继续充钱变强？？作为一个理智的我，肯定得想想是哪里有问题。现在910便利网的模块有好几个，全都丢在同一个Tomcat里边。

其实有些模块的访问是很低的(比如后台管理)，那我可不可以这样做：将每个模块**抽取独立**出来，访问量大的模块用好的服务器装着，没啥人访问的模块用差的服务器装着。这样的好处是：一、**资源合理利用了**(没人访问的模块用性能差的服务器，访问量大的模块**单独提升性能**就好了)。二、**耦合度降低了**：每个模块独立出来，各干各的事(专业的人做专业的事)，便于扩展

特点：

- 将910便利网的**功能拆分，模块之间独立**，在使用的时候再将这些**独立的模块组合起来**就是一个系统了。

好处：

- 模块之间独立，各做各的事，**便于扩展，复用性高**
- **高吞吐量。**某个任务需要一个机器运行10个小时，将该任务用10台机器的分布式跑(将这个任务拆分成10个小任务)，可能2个小时就跑完了

**分布式**：一个业务分拆多个子业务，部署在不同的服务器上(不同的服务器，运行不同的代码，为了同一个目的)

## 2.3集群/分布式

集群和分布式并不冲突，可以有**分布式集群**

现在3y的公司规模变大了，有5个小伙子写Java，4个小伙子写前端，2个小伙子做测试，1个小伙子做DBA。

- Java，前端，测试，DBA的关系看作是分布式的
- 5个Java看作是集群的(前端，测试同理)...

## 2.4分布式/微服务/SOA

其实我认为分布式/微服务/SOA这三个概念是差不多的，了解了其中的一个，然后将自己的理解往上面套就好了。**没必要细分每个的具体概念**~~(当然了，我很期待有大佬可以在评论区留言说下自己的看法哈)

参考资料：

- 分布式与集群的区别是什么？https://www.zhihu.com/question/20004877
- 分布式、集群、微服务、SOA 之间的区别 https://blog.csdn.net/heatdeath/article/details/79038795

# 3 CAP理论

从上面所讲的分布式概念我们已经知道，分布式简单理解就是：**一个业务分拆多个子业务，部署在不同的服务器上**

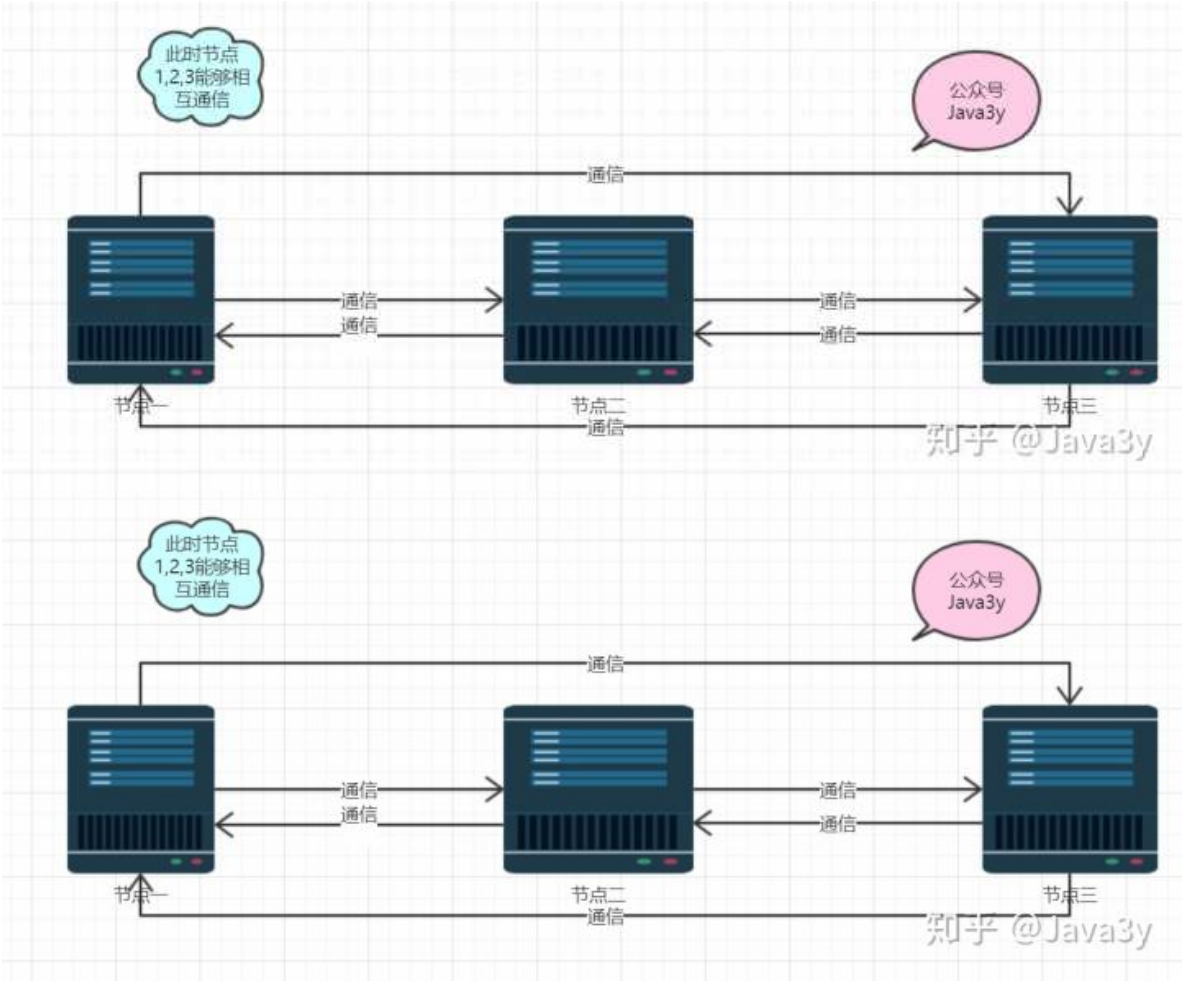
- 一般来说，一个子业务我们称为**节点。**

如果你接触过一些分布式的基础概念，那肯定会听过CAP这个理论。就比如说：你学了MySQL的InnoDB存储引擎相关知识，你肯定听过ACID！

首先，我们来看一下CAP分别代表的是什么意思：

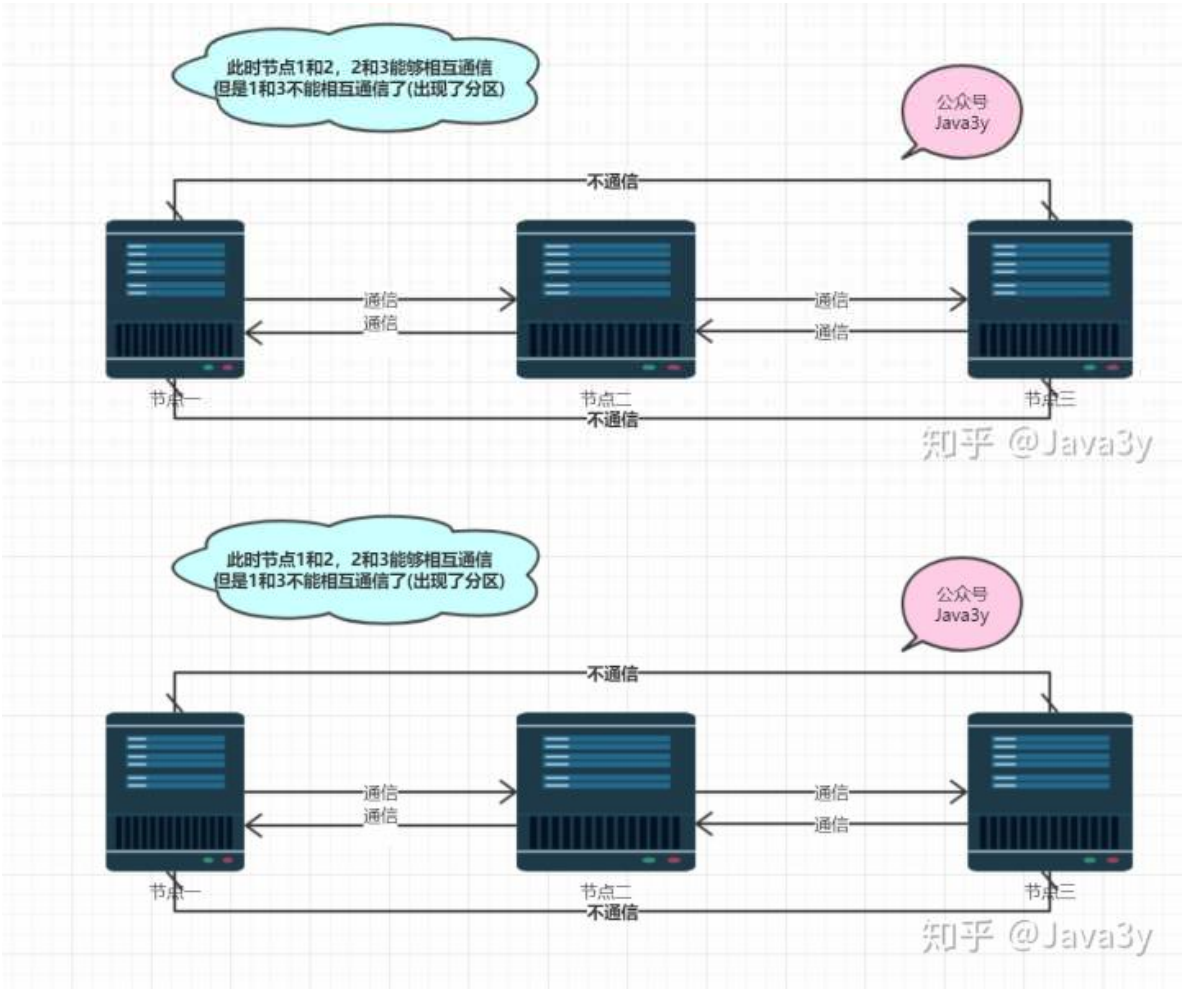
- C：数据一致性(consistency)
  - **所有**节点拥有数据的最新版本
- A：可用性(availability)
  - 数据具备高可用性
- P：分区容错性(partition-tolerance)
  - **容忍网络出现分区**，分区之间网络不可达。

下面有三个节点(它们是集群的)，此时三个节点都能够相互通信：

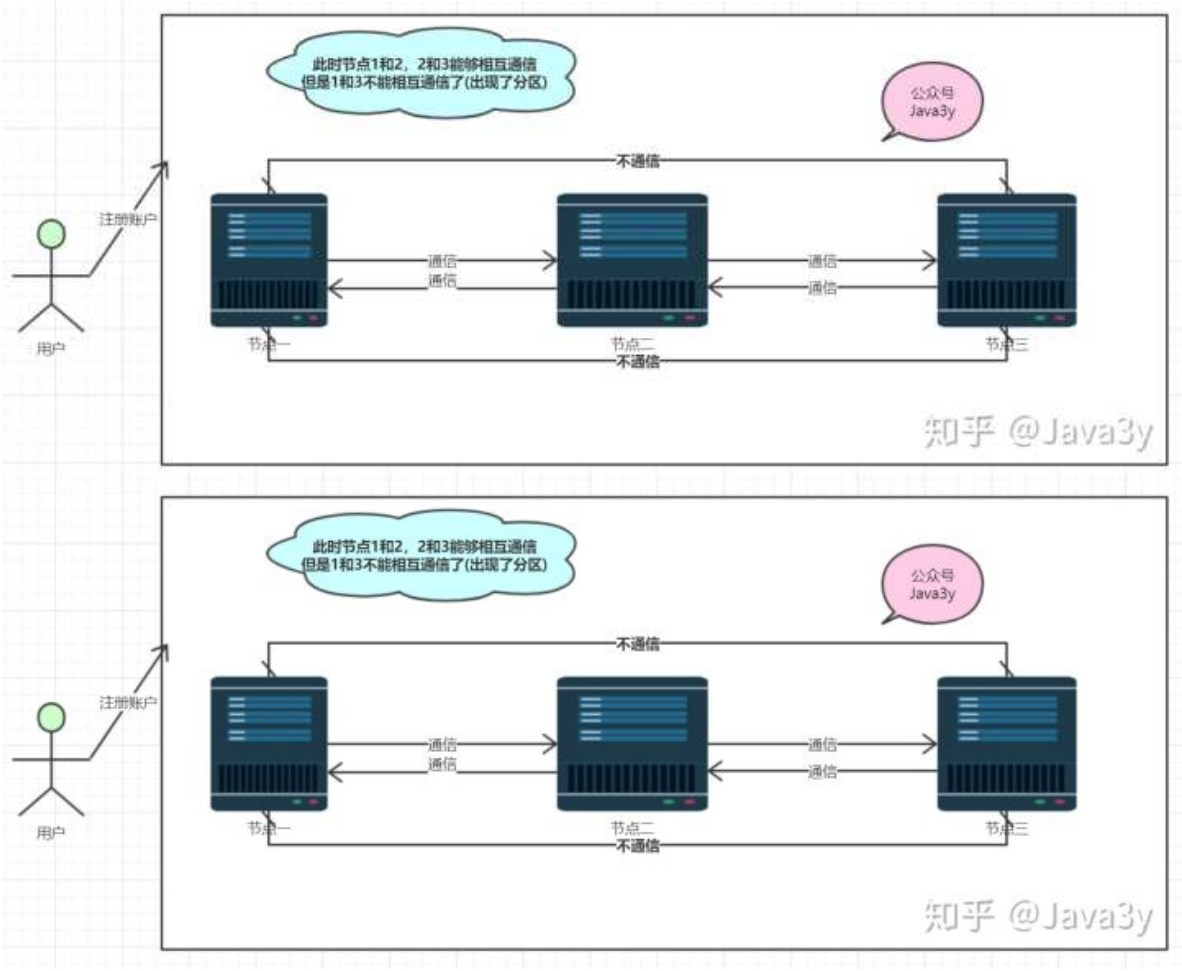



由于我们的系统是分布式的，节点之间的通信是通过网络来进行的。**只要是分布式系统**，那很有可能会出现一种情况：因为一些**故障**，使得有些**节点之间不连通**了，整个网络就分成了**几块区域。**

- 数据就散布在了这些不连通的区域中，这就叫**分区**




现在出现了网络分区后，此时有一个请求过来了，想要注册一个账户。




此时我们**节点一和节点三是不可通信的**，这就有了抉择：

- 如果**允许**当前用户注册一个账户，此时注册的记录数据只会在节点一和节点二或者节点二和节点三**同步**，因为节点一和节点三的记录不能同步的。
    - 这种情况其实就是选择了可用性(availability)，抛弃了数据一致性(consistency)
- 如果**不允许**当前用户注册一个账户(就是要**等到**节点一和节点三恢复通信)。节点一和节点三一旦恢复通信，我们就可以**保证节点拥有的数据是最新版本。**
    - 这种情况其实就是抛弃了可用性(availability)，选择了数据一致性(consistency)

## 3.1再次梳理一下CAP理论

一般我们说的分布式系统，P：分区容错性(partition-tolerance)这个是**必需**的，这是客观存在的。

CAP是无法完全兼顾的，从上面的例子也可以看出，我们可以选AP，也可以选CP。但是，要**注意的是**：不是说选了AP，C就完全抛弃了。不是说选了CP，A就完全抛弃了！

在CAP理论中，**C所表示的一致性是强一致性**(每个节点的数据都是最新版本)，其实一致性还有其他级别的：

- 弱一致性：弱一致性是相对于强一致性而言，它不保证总能得到最新的值；
- 最终一致性(eventual consistency)：放宽对时间的要求，在被调完成操作响应后的某个时间点，被调多个节点的数据最终达成一致

可用性的值域可以定义成**0到100%的连续区间。**

| 可用性分类 | 可用水平（%） | 年可容忍停机时间 |
| --- | --- | --- |
| 容错可用性 | 99.9999 | <1 min |
| 极高可用性 | 99.999 | <5 min |
| 具有故障自动恢复能力的可用性 | 99.99 | <53 min |
| 高可用性 | 99.9 | <8.8h |
| 商品可用性 | 99 | <43.8 min |
| 可用性分类 | 可用水平（%） | 年可容忍停机时间 |
| 容错可用性 | 99.9999 | <1 min |
| 极高可用性 | 99.999 | <5 min |
| 具有故障自动恢复能力的可用性 | 99.99 | <53 min |
| 高可用性 | 99.9 | <8.8h |
| 商品可用性 | 99 | <43.8 min |



所以，**CAP理论定义的其实是在容忍网络分区的条件下，“强一致性”和“极致可用性”无法同时达到。**

参考资料：

- CAP理论中的P到底是个什么意思？https://www.zhihu.com/question/54105974
- 浅谈分布式系统的基本问题：可用性与一致性：https://m.aliyun.com/yunqi/articles/2709
- 分布式系统的CAP理论：http://www.hollischuang.com/archives/666
- 为什么CAP理论在舍弃P的情况下，可以有完美的CA？https://www.zhihu.com/question/28587819
- 不懂点CAP理论，你好意思说你是做分布式的吗？http://www.yunweipai.com/archives/8432.html

扩展阅读：

- 浅谈分布式事务：https://m.aliyun.com/yunqi/articles/230242

# 4 SpringCloud就是这么简单

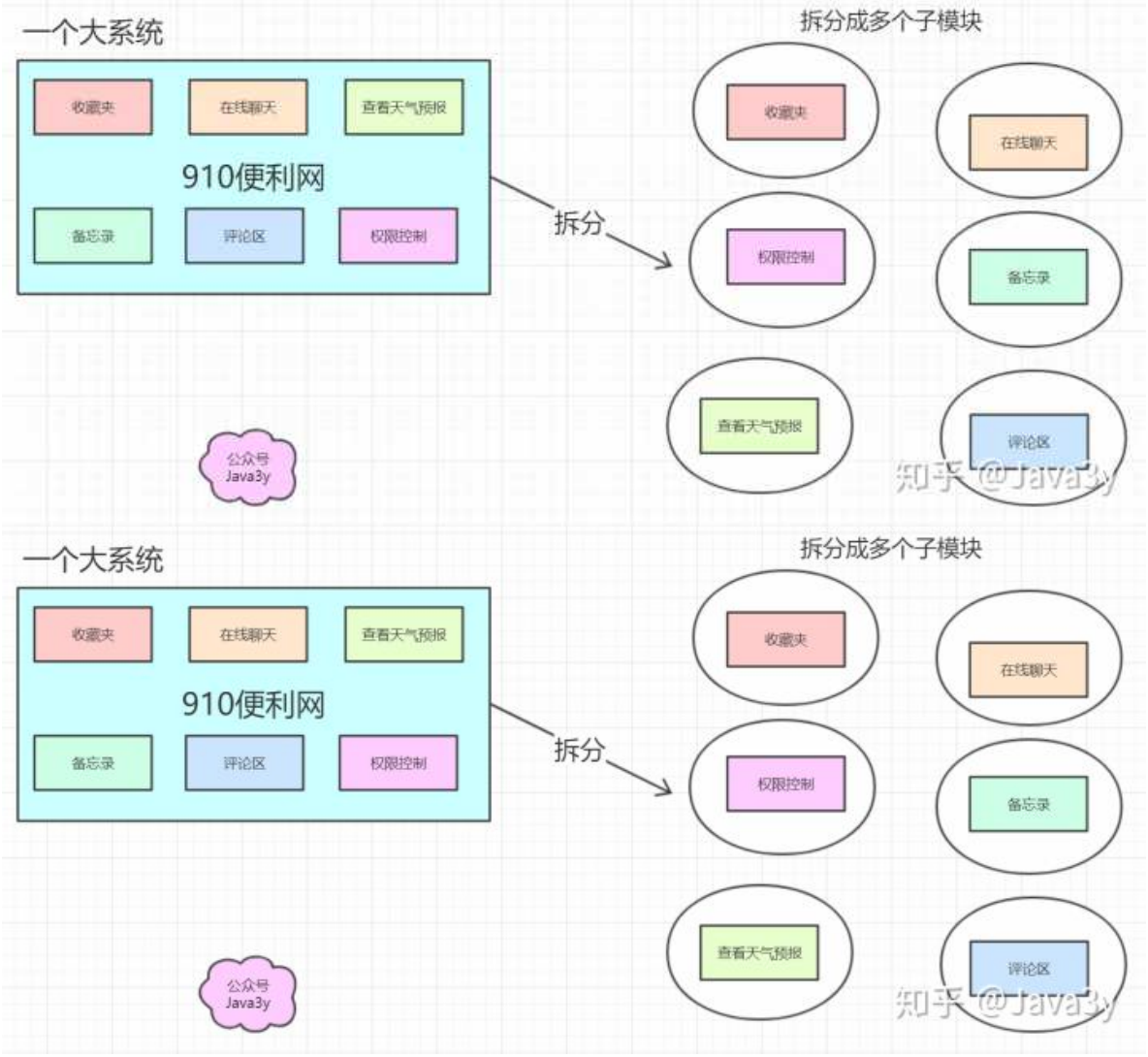
相信大家读到这里，对分布式/微服务已经有一定的了解了，其实单从概念来说，是非常容易理解的。只是很可能被它的名字给唬住了。

下面我就来讲讲SpringCloud**最基础**的知识~

## 4.1为什么需要SpringCloud?

前面也讲了，从分布式/微服务的角度而言：就是把我们一**大**的项目，**分解**成多个**小**的模块。这些小的模块组合起来，完成功能。

举个可能不太恰当的例子(现实可能不会这么拆分，但意思到位就好了)：




拆分出多个模块以后，就会出现**各种各样**的问题，而SpringCloud提供了**一整套**的解决方案！

- 注：这些模块是**独立**成一个子系统的(不同主机)。

SpringCloud的**基础功能**：

- 服务治理： Spring Cloud Eureka
- 客户端负载均衡： Spring Cloud Ribbon
- 服务容错保护： Spring Cloud Hystrix
- 声明式服务调用： Spring Cloud Feign
- API网关服务：Spring Cloud Zuul
- 分布式配置中心： Spring Cloud Config

SpringCloud的高级功能(本文不讲)：

- 消息总线： Spring Cloud Bus
- 消息驱动的微服务： Spring Cloud Stream
- 分布式服务跟踪： Spring Cloud Sleuth

# 5 引出Eureka

那会出现什么问题呢？？首当其冲的就是子系统之间的**通讯**问题。子系统与子系统之间不是在同一个环境下，那就需要**远程调用**。远程调用可能就会想到httpClient，WebService等等这些技术来实现。

既然是远程调用，就必须知道ip地址，我们可能有以下的场景。

- 功能实现一：A服务需要调用B服务
    - 在A服务的代码里面调用B服务，**显式通过IP地址调用**：`http://123.123.123.123:8888/java3y/3`
- 功能实现二：A服务调用B服务，B服务调用C服务，C服务调用D服务
    - 在A服务的代码里面调用B服务，显式通过IP地址调用：`http://123.123.123.123:8888/java3y/3`，(同样地)B->C，C->D
- 功能实现三：D服务调用B服务，B服务调用C服务
    - 在D服务的代码里面调用B服务，显式通过IP地址调用：`http://123.123.123.123:8888/java3y/3`，(同样地)B->C

- .....等等等等

万一，我们**B服务的IP地址变了**，想想会出现什么问题：A服务,D服务(等等)需要**手动更新**B服务的地址

- 在服务多的情况下，手动来维护这些静态配置就是噩梦！

> 为了解决微服务架构中的**服务实例维护问题(ip地址)**， 产生了大量的**服务治理**框架和产品。 这些框架和产品的实现都围绕着服务注册与服务发现机制来完成对微服务应用实例的**自动化管理**。

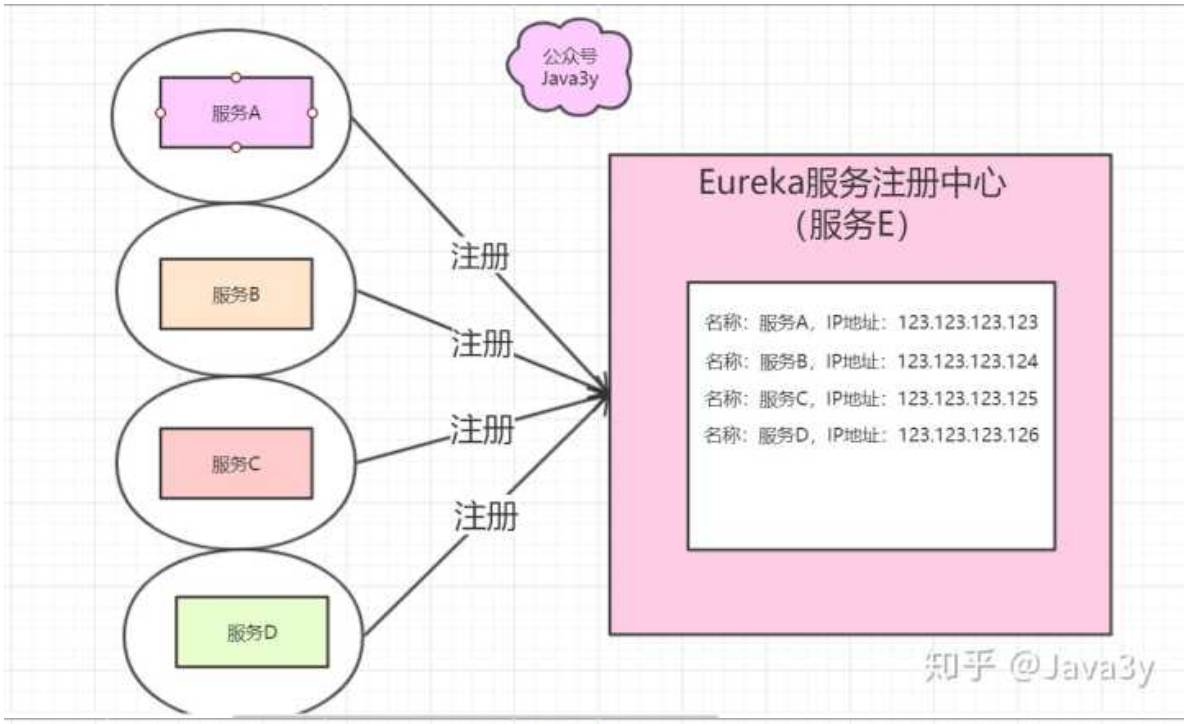
在SpringCloud中我们的服务治理框架一般使用的就是Eureka。

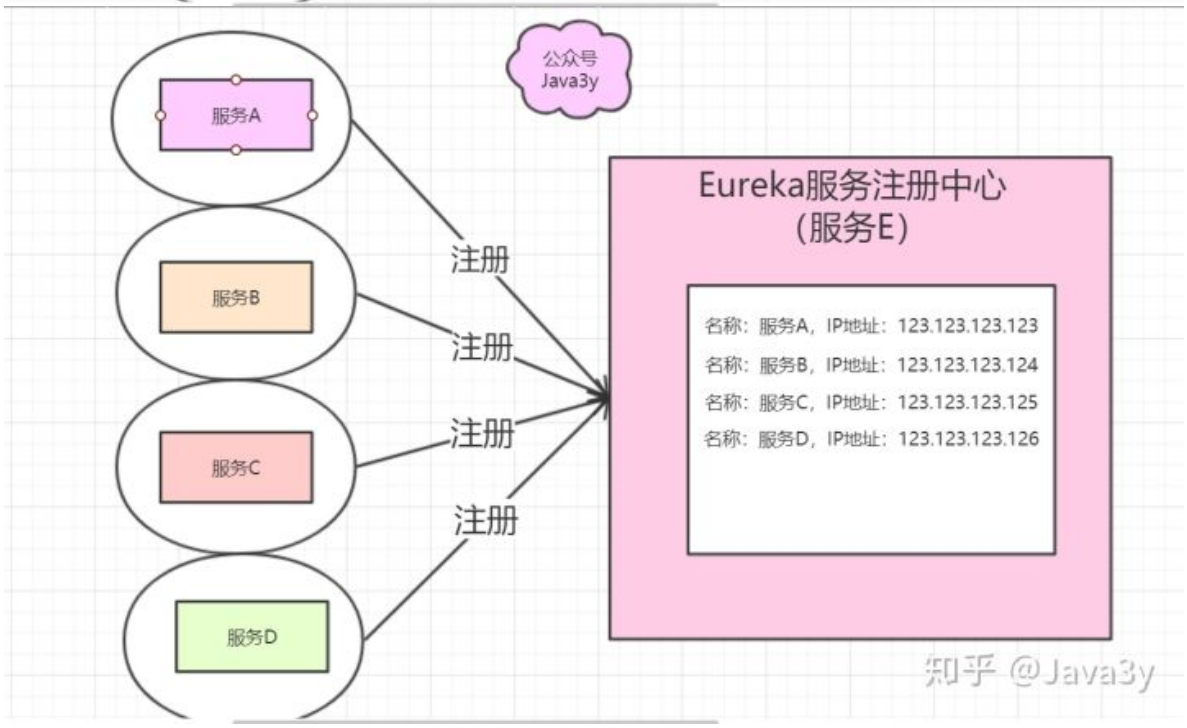
我们的问题：

- 现在有A、B、C、D四个服务，它们之间会互相调用(而且IP地址很可能会发生变化)，一旦某个服务的IP地址变了，那服务中的代码要跟着变，手动维护这些静态配置(IP)非常麻烦！

Eureka是这样解决上面所说的情况的：

- 创建一个E服务，将A、B、C、D四个服务的信息都**注册**到E服务上，E服务维护这些已经注册进来的信息




A、B、C、D四个服务都可以**拿到**Eureka(服务E)那份**注册清单**。A、B、C、D四个服务互相调用不再通过具体的IP地址，而是**通过服务名来调用**!

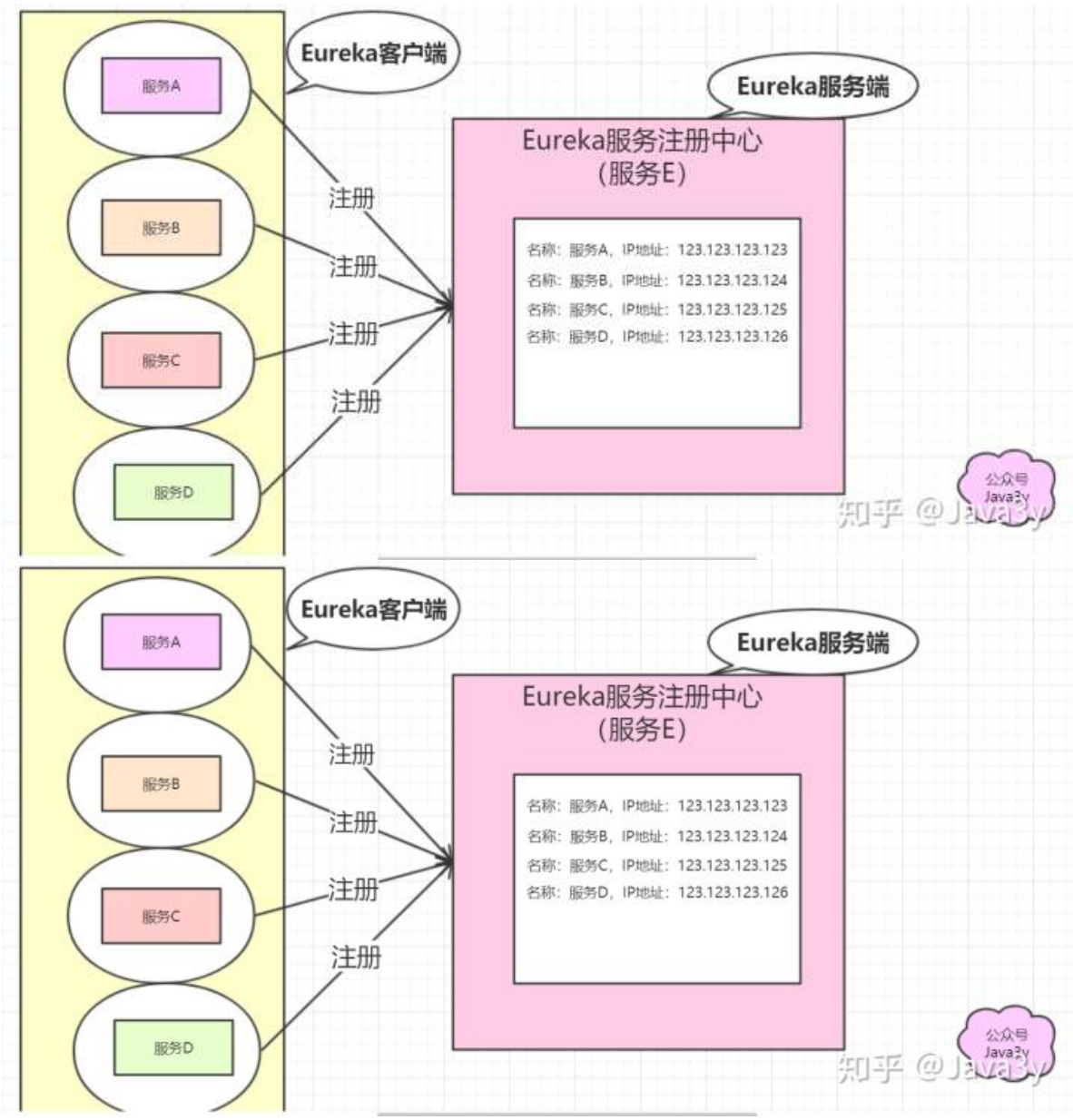
- 拿到注册清单--->注册清单上有服务名--->自然就能够拿到服务具体的位置了(IP)。
- 其实简单来说就是：代码中通过**服务名找到对应的IP地址**(IP地址会变，但服务名一般不会变)

| 服务名 | 位置 |
| --- | --- |
| 服务 A | 192.168.0.100:8000、192.168.0.101:8000 |
| 服务 B | 192.168.0.100:9000、192.168.0.101:9000、192.168.0.102:9000 |


| 服务名 | 位置 |
| --- | --- |
| 服务 A | 192.168.0.100:8000、192.168.0.101:8000 |
| 服务 B | 192.168.0.100:9000、192.168.0.101:9000、192.168.0.102:9000 |


## 5.1Eureka细节

Eureka专门用于给其他服务注册的称为Eureka Server(服务注册中心)，其余注册到Eureka Server的服务称为Eureka Client。




在Eureka Server一般我们会这样配置：

```
register-with-eureka: false     #false表示不向注册中心注册自己。
    fetch-registry: false     #false表示自己端就是注册中心，我的职责就是维护服务实例，并不需要去检索服务
```

Eureka Client**分为服务提供者和服务消费者。**

- 但很可能，某服务**既是服务提供者又是服务消费者。**

如果在网上看到SpringCloud的**某个服务配置没有"注册"到Eureka-Server也不用过于惊讶**(但是它是可以获取Eureka服务清单的)

- 很可能只是作者把该服务认作为**单纯的服务消费者**，单纯的服务消费者无需对外提供服务，也就无须注册到Eureka中了

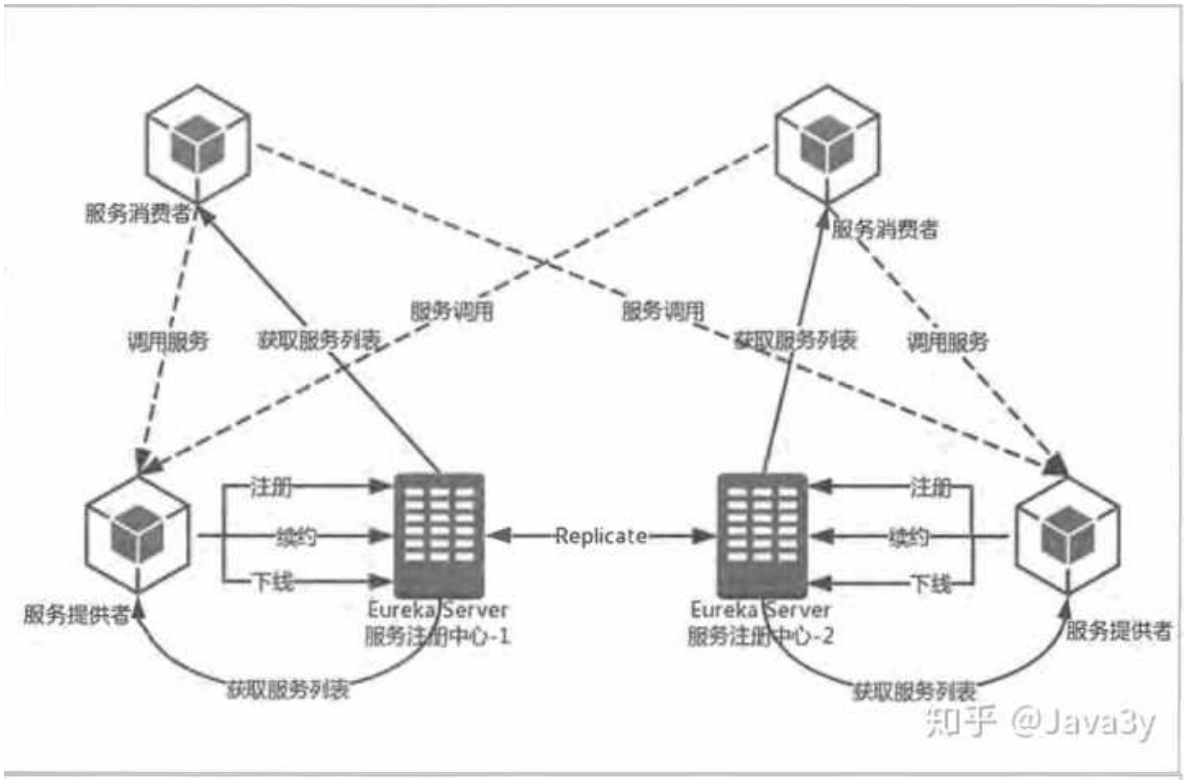
```
eureka:
  client:
    register-with-eureka: false  # 当前微服务不注册到eureka中(消费端)
    service-url:
      defaultZone:
http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/
```

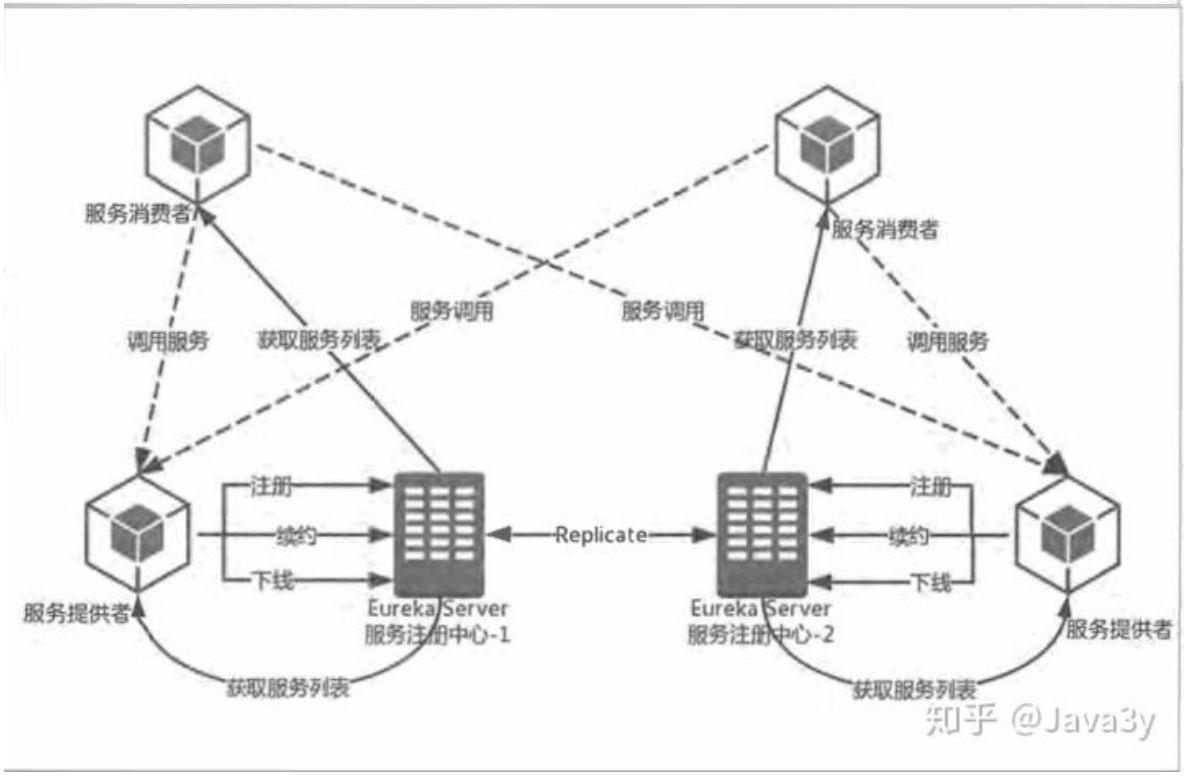
下面是Eureka的治理机制：

- 服务提供者
    - **服务注册：启动的时候会通过发送REST请求的方式将自己注册到Eureka Server上**，同时带上了自身服务的一些元数据信息。
    - **服务续约：**在注册完服务之后，**服务提供者会维护一个心跳**用来持续告诉Eureka Server: "我还活着 ”、
    - **服务下线：当服务实例进行正常的关闭操作时，它会触发一个服务下线的REST请求**给Eureka Server, 告诉服务注册中心：“我要下线了 ”。
- 服务消费者
    - **获取服务：当我们启动服务消费者**的时候，它会发送一个REST请求给服务注册中心，来获取上面注册的服务清单
    - **服务调用：服务消费者在获取服务清单后，通过服务名**可以获得具体提供服务的实例名和该实例的元数据信息。在进行服务调用的时候，**优先访问同处一个Zone中的服务提供方。**

- Eureka Server(服务注册中心)：
    - **失效剔除：**默认每隔一段时间（默认为60秒） 将当前清单中超时（默认为90秒）**没有续约的服务剔除出去。**
    - **自我保护：**。EurekaServer 在运行期间，会统计心跳失败的比例在15分钟之内是否低于85%(通常由于网络不稳定导致)。 Eureka Server会将当前的**实例注册信息保护起来**， 让这些实例不会过期，尽可能**保护这些注册信息。**

最后，我们就有了这张图：




举个例子：

- 3y跟女朋友去东站的东方宝泰逛街，但不知道东方宝泰有什么好玩的。于是就去**物业**搜了一下**东方宝泰商户清单**，发现一楼有优衣库，二楼有星巴克，三楼有麦当劳。
- 在优衣库旁边，有新开张的KFC，在墙壁打上了很大的标识“欢迎KFC**入驻**东方宝泰”。
- 商家们需要定时**交物业费**给物业。
- **物业维持**东方宝泰的稳定性。如果某个商家不想在东方宝泰运营了，告诉了物业。物业自然就会将其在东方宝泰商户清单去除。

优秀博文：

- Spring Cloud Eureka详解：https://blog.csdn.net/sunhuiliang85/article/details/76222517

- 《Spring Cloud Netflix》 -- 服务注册和服务发现-Eureka 的使用：https://zhuanlan.zhihu.com/p/26472547
- 微服务架构：Eureka参数配置项详解：https://www.cnblogs.com/fangfuhai/p/7070325.html

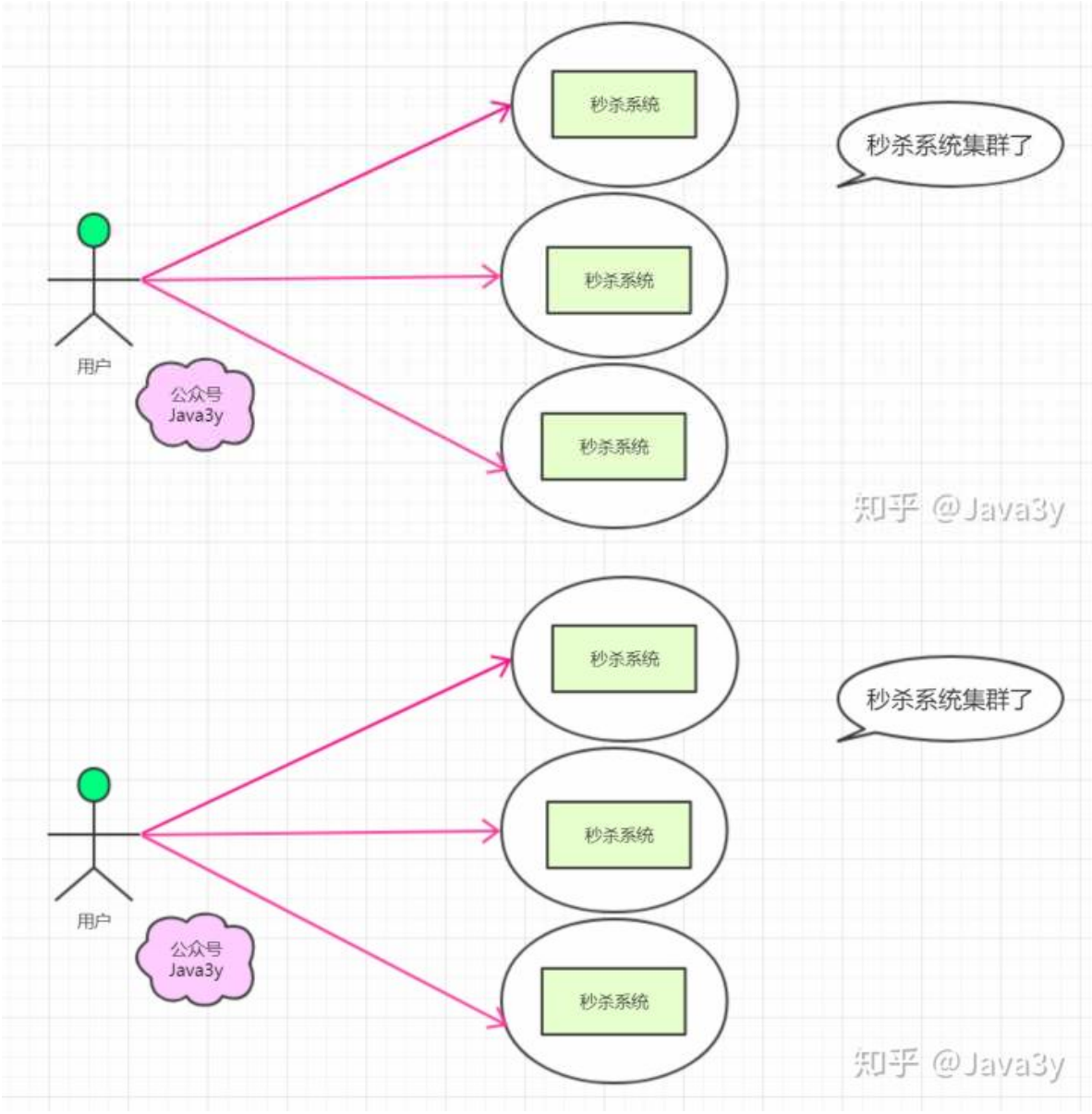
# 6 引出RestTemplate和Ribbon

通过Eureka服务治理框架，我们可以通过服务名来获取具体的服务实例的位置了(IP)。一般在使用SpringCloud的时候**不需要自己手动创建**HttpClient来进行远程调用。

可以使用Spring封装好的**RestTemplate**工具类，使用起来很简单：

```
// 传统的方式，直接显示写死IP是不好的!
    //private static final String REST_URL_PREFIX = "http://localhost:8001";

    // 服务实例名
    private static final String REST_URL_PREFIX = "http://MICROSERVICECLOUD-
DEPT";

    /**
     * 使用 使用restTemplate访问restful接口非常的简单粗暴无脑。 (url, requestMap,
     * ResponseBean.class)这三个参数分别代表 REST请求地址、请求参数、HTTP响应转换被转换成
的对象类型。
     */
    @Autowired
    private RestTemplate restTemplate;

    @RequestMapping(value = "/consumer/dept/add")
    public boolean add(Dept dept) {
        return restTemplate.postForObject(REST_URL_PREFIX + "/dept/add", dept,
Boolean.class);
    }
```

为了实现服务的**高可用**，我们可以将**服务提供者集群。**比如说，现在一个秒杀系统设计出来了，准备上线了。在11月11号时为了能够支持高并发，我们开多台机器来支持并发量。



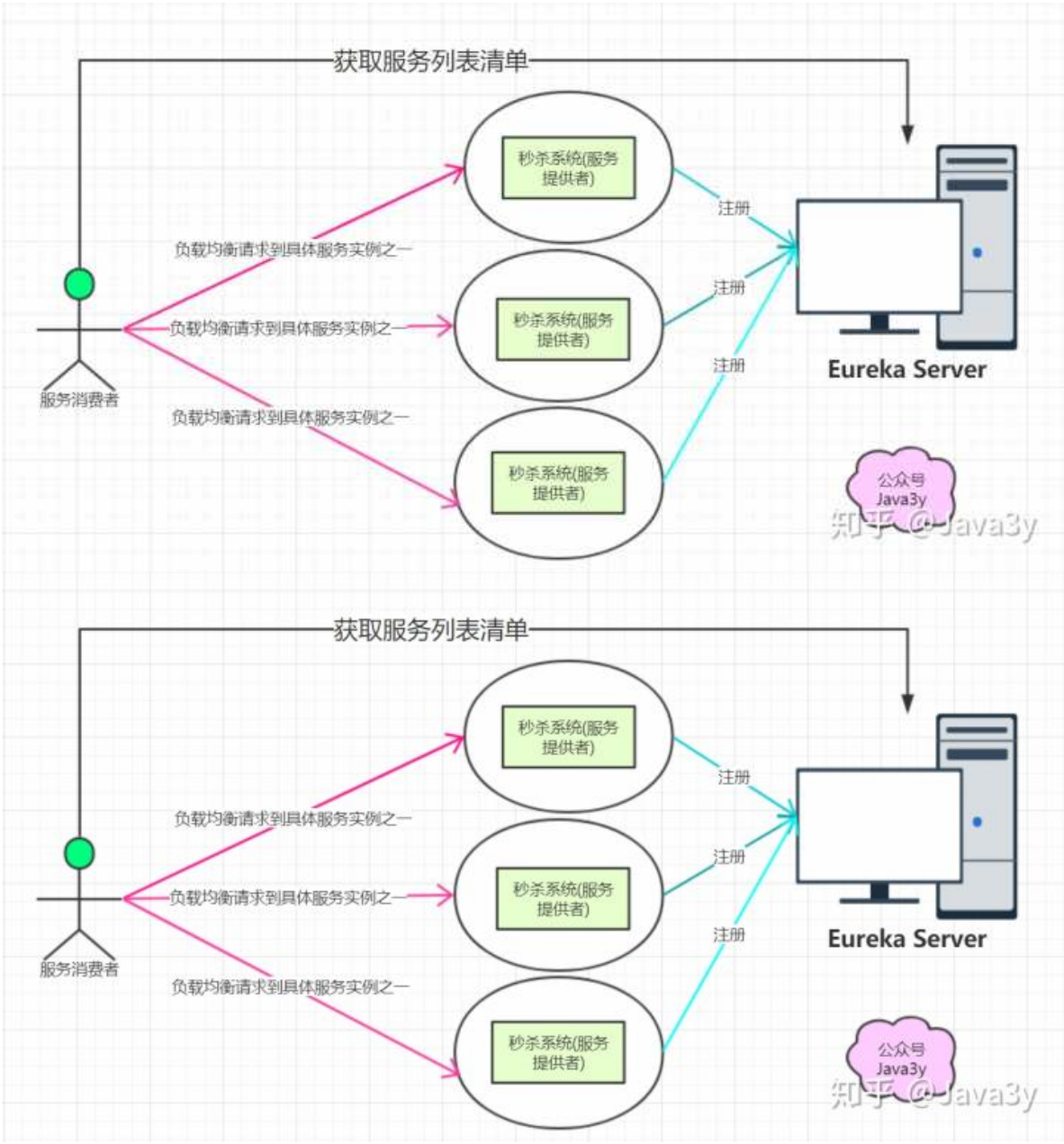

现在想要这三个秒杀系统**合理摊分**用户的请求(专业来说就是负载均衡)，可能你会想到nginx。

其实SpringCloud也支持的负载均衡功能，只不过它是**客户端的负载均衡**，这个功能实现就是Ribbon！

负载均衡又区分了两种类型：

- 客户端负载均衡(Ribbon)
    - 服务实例的**清单在客户端**，客户端进行负载均衡算法分配。
    - (从上面的知识我们已经知道了：客户端可以从Eureka Server中得到一份服务清单，在发送请求时通过负载均衡算法，**在多个服务器之间选择一个进行访问**)
- 服务端负载均衡(Nginx)
    - 服务实例的**清单在服务端**，服务器进行负载均衡算法分配

所以，我们的图可以画成这样：




## 6.1Ribbon细节

Ribbon是支持负载均衡，默认的负载均衡策略是轮询，我们也是可以根据自己实际的需求自定义负载均衡策略的。

```
@Configuration
public class MySelfRule
{
    @Bean
    public IRule myRule()
    {
        //return new RandomRule();// Ribbon默认是轮询，我自定义为随机
        //return new RoundRobinRule();// Ribbon默认是轮询，我自定义为随机

        return new RandomRule_ZY();// 我自定义为每台机器5次
    }
}
```

实现起来也很简单：继承AbstractLoadBalancerRule类，重写`public Server choose(ILoadBalancer lb, Object key)`即可。

SpringCloud 在CAP理论是选择了AP的，在Ribbon中还可以配置**重试机制**的(有兴趣的同学可以去搜搜)~

举个例子：

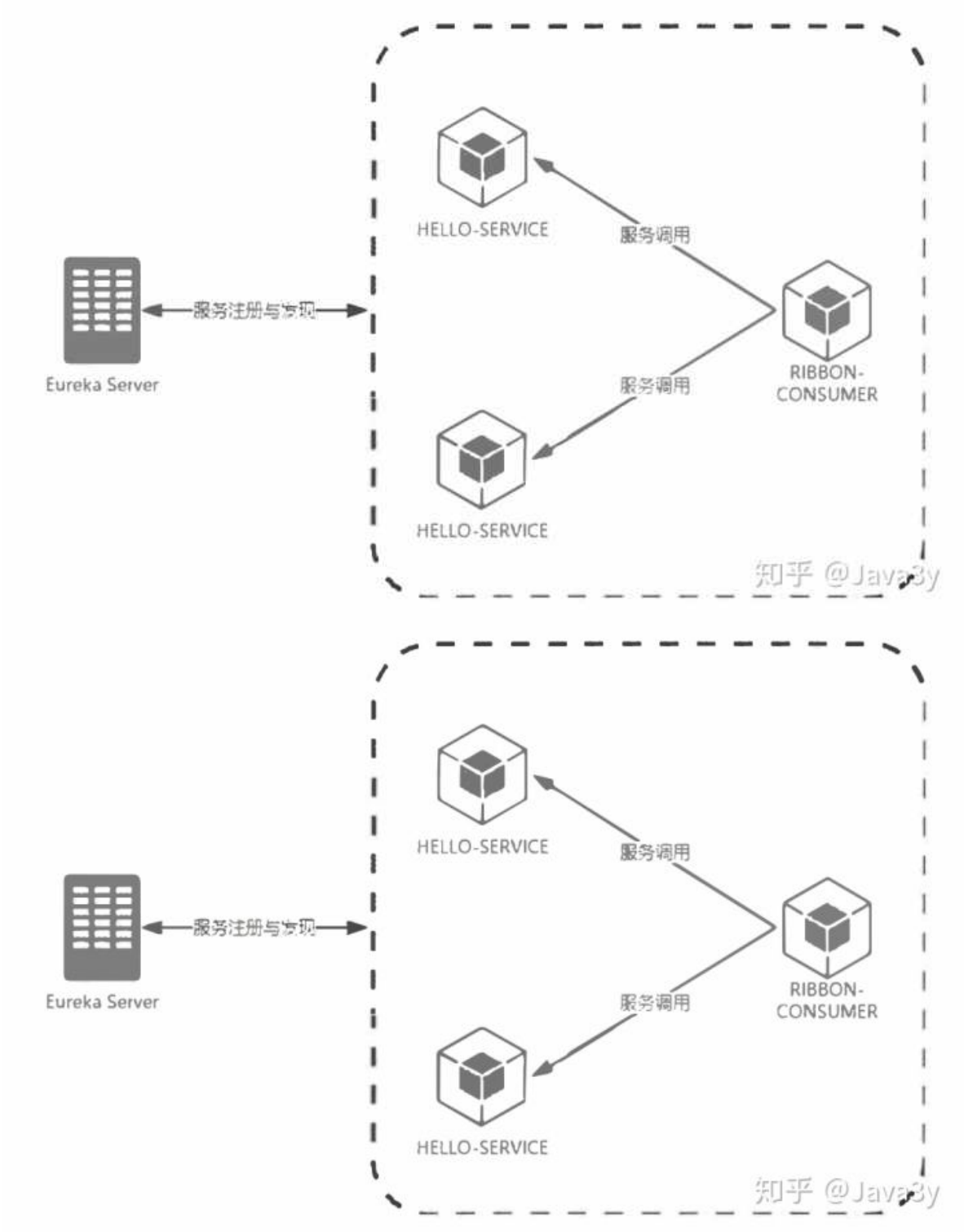
- 3y跟女朋友过了几个月，又去东方宝泰了。由于记性不好，又去物业那弄了一份东方宝泰商户清单。
- 这次看到东方宝泰又开了一间麦当劳，一间在二楼，一间在三楼。原来生意太好了，为了能提高用户体验，在二楼**多开了一间麦当劳**。
- 这时，3y问女朋友：“去哪间麦当劳比较好？要不我们抛硬币决定？”3y女朋友说：”你是不是傻，肯定哪间近去哪间啊“

优秀博文：

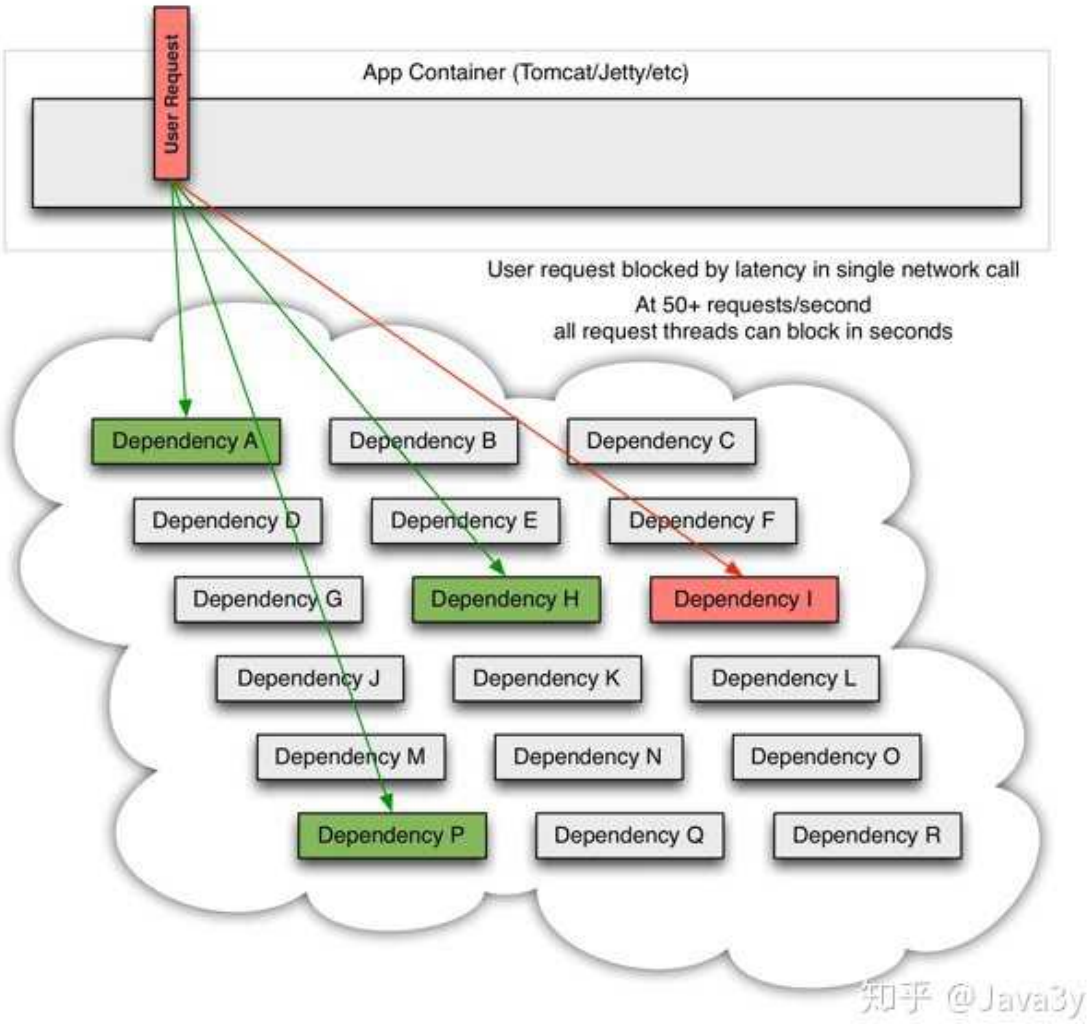
- 撸一撸Spring Cloud Ribbon的原理-负载均衡策略：https://www.cnblogs.com/kongxianghai/p/8477781.html

# 7 引出Hystrix

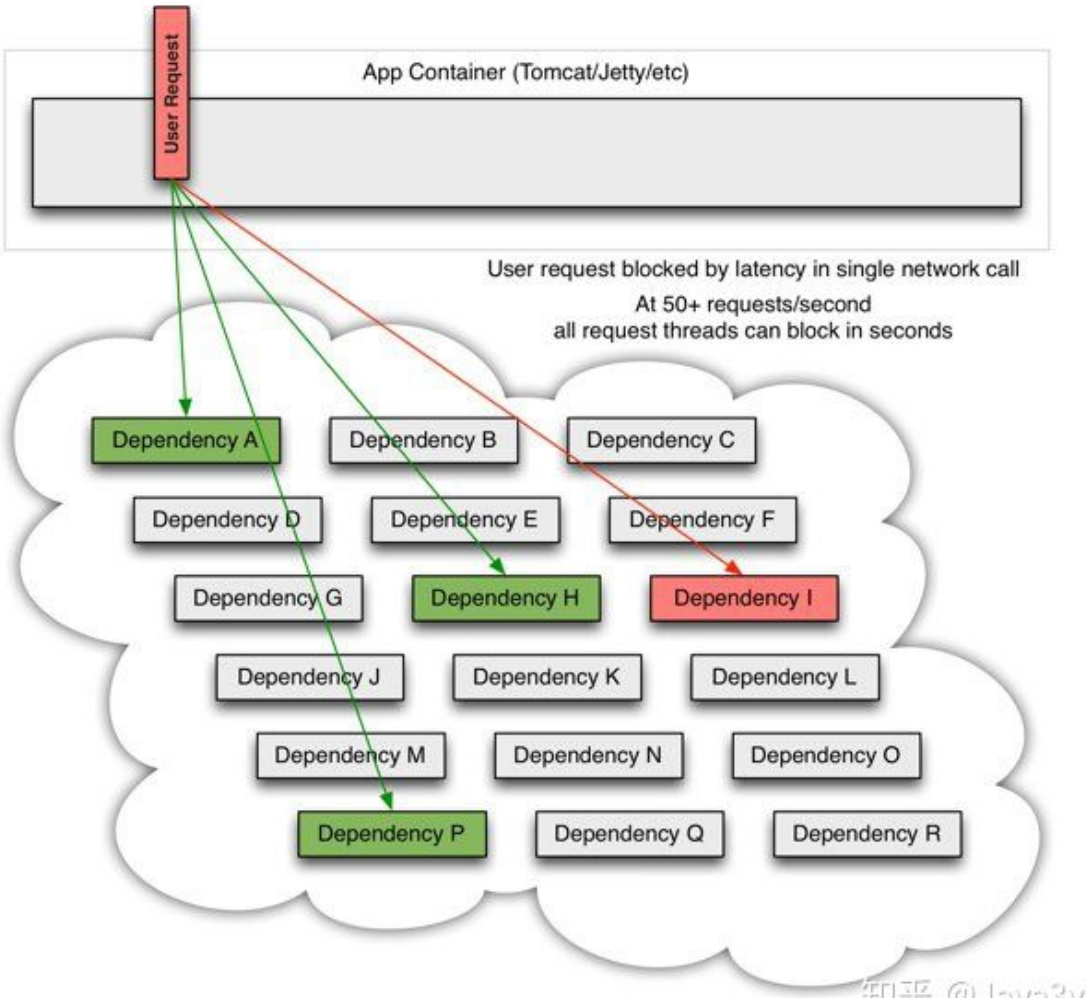
到目前为止，我们的服务看起来好像挺好的了：能够根据服务名来远程调用其他的服务，可以实现客户端的负载均衡。



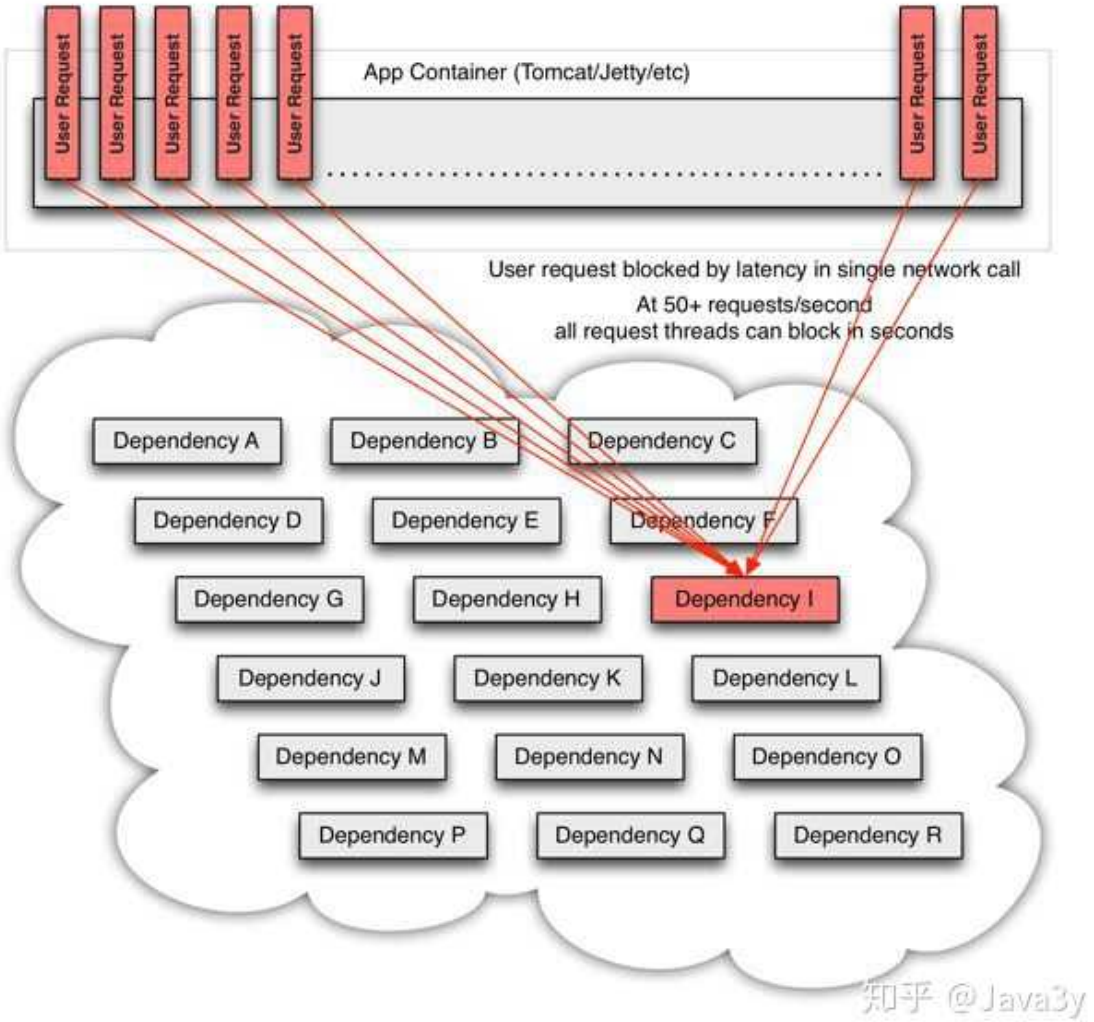

但是，如果我们在**调用多个远程服务时，某个服务出现延迟**，会怎么样？？


User Request
App Container (Tomcat/Jetty/etc)
User request blocked by latency in single network call
At 50+ requests/second
all request threads can block in seconds
Dependency A
Dependency B
Dependency C
Dependency D
Dependency E
Dependency F
Dependency G
Dependency H
Dependency I
Dependency J
Dependency K
Dependency L
Dependency M
Dependency N
Dependency O
Dependency P
Dependency Q
Dependency R

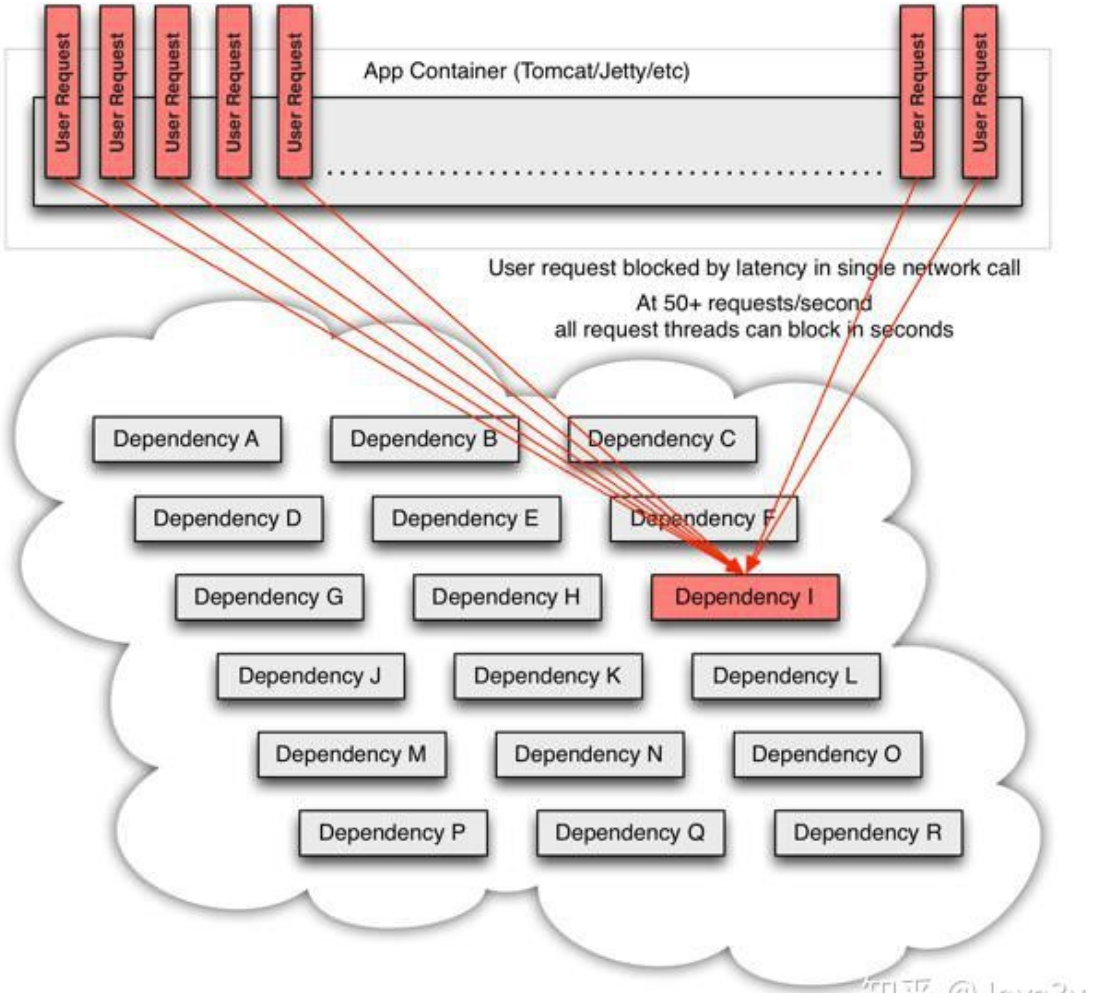



User Request
App Container (Tomcat/Jetty/etc)
User request blocked by latency in single network call
At 50+ requests/second
all request threads can block in seconds
Dependency A
Dependency B
Dependency C
Dependency D
Dependency E
Dependency F
Dependency G
Dependency H
Dependency I
Dependency J
Dependency K
Dependency L
Dependency M
Dependency N
Dependency O
Dependency P
Dependency Q
Dependency R

在**高并发**的情况下，由于单个服务的延迟，可能导致**所有的请求都处于延迟状态**，甚至在几秒钟就使服务处于负载饱和的状态，资源耗尽，直到不可用，最终导致这个分布式系统都不可用，这就是“雪崩”。


User Request
User Request
User Request
User Request
User Request
App Container (Tomcat/Jetty/etc)
User Request
User Request
User request blocked by latency in single network call
At 50+ requests/second
all request threads can block in seconds
Dependency A
Dependency B
Dependency C
Dependency D
Dependency E
Dependency F
Dependency G
Dependency H
Dependency I
Dependency J
Dependency K
Dependency L
Dependency M
Dependency N
Dependency O
Dependency P
Dependency Q
Dependency R




User Request
User Request
User Request
User Request
User Request
App Container (Tomcat/Jetty/etc)
User Request
User Request
User request blocked by latency in single network call
At 50+ requests/second
all request threads can block in seconds
Dependency A
Dependency B
Dependency C
Dependency D
Dependency E
Dependency F
Dependency G
Dependency H
Dependency I
Dependency J
Dependency K
Dependency L
Dependency M
Dependency N
Dependency O
Dependency P
Dependency Q
Dependency R

针对上述问题， Spring Cloud Hystrix实现了**断路器、线程隔离**等一系列服务保护功能。

- Fallback(失败快速返回)：当某个服务单元发生故障（类似用电器发生短路）之后，通过断路器的故障监控（类似熔断保险丝），**向调用方返回一个错误响应，而不是长时间的等待。**这样就不会使得线程因调用故障服务被长时间占用不释放，**避免**了故障在分布式系统中的**蔓延。**
- 资源/依赖隔离(线程池隔离)：它会为**每一个依赖服务创建一个独立的线程池**，这样就算某个依赖服务出现延迟过高的情况，也只是对该依赖服务的调用产生影响，而**不会拖慢其他的依赖服务。**

Hystrix提供几个熔断关键参数：`滑动窗口大小（20）、 熔断器开关间隔（5s）、错误率（50%）`

- 每当20个请求中，有50%失败时，熔断器就会打开，此时再调用此服务，将会**直接返回失败**，不再调远程服务。
- 直到5s钟之后，重新检测该触发条件，**判断是否把熔断器关闭，或者继续打开。**

Hystrix还有请求合并、请求缓存这样强大的功能，在此我就不具体说明了，有兴趣的同学可继续深入学习~

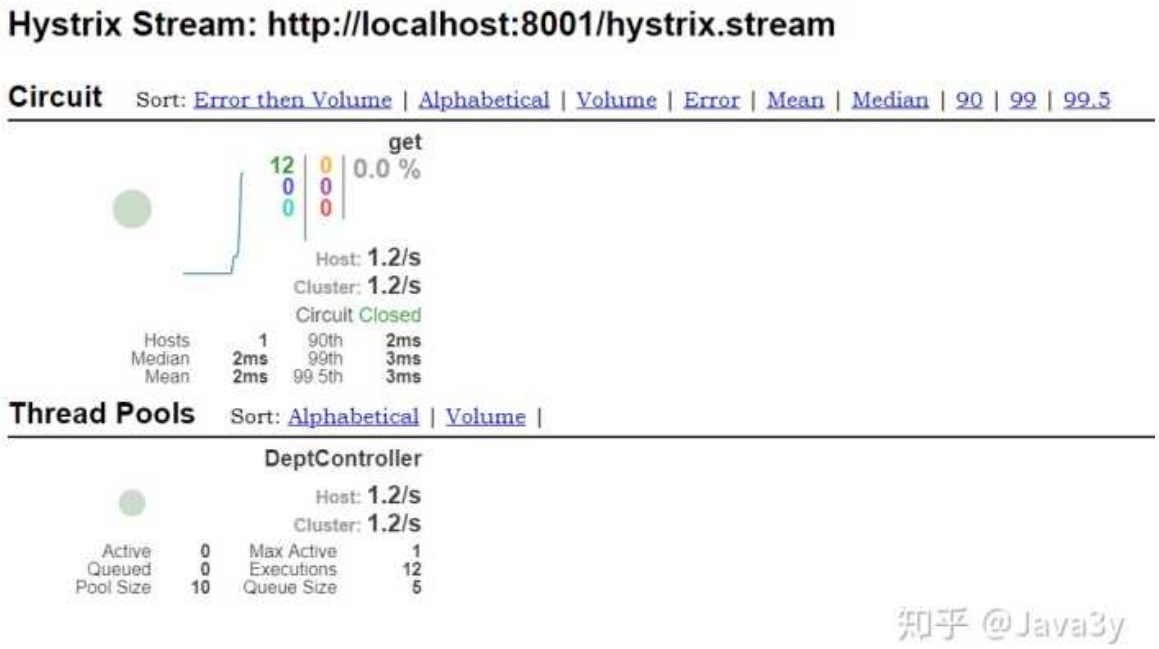
## 7.1Hystrix仪表盘

Hystrix仪表盘：它主要用来**实时监控Hystrix的各项指标信息。**通过Hystrix Dashboard反馈的实时信息，可以帮助我们快速发现系统中存在的问题，从而及时地采取应对措施。

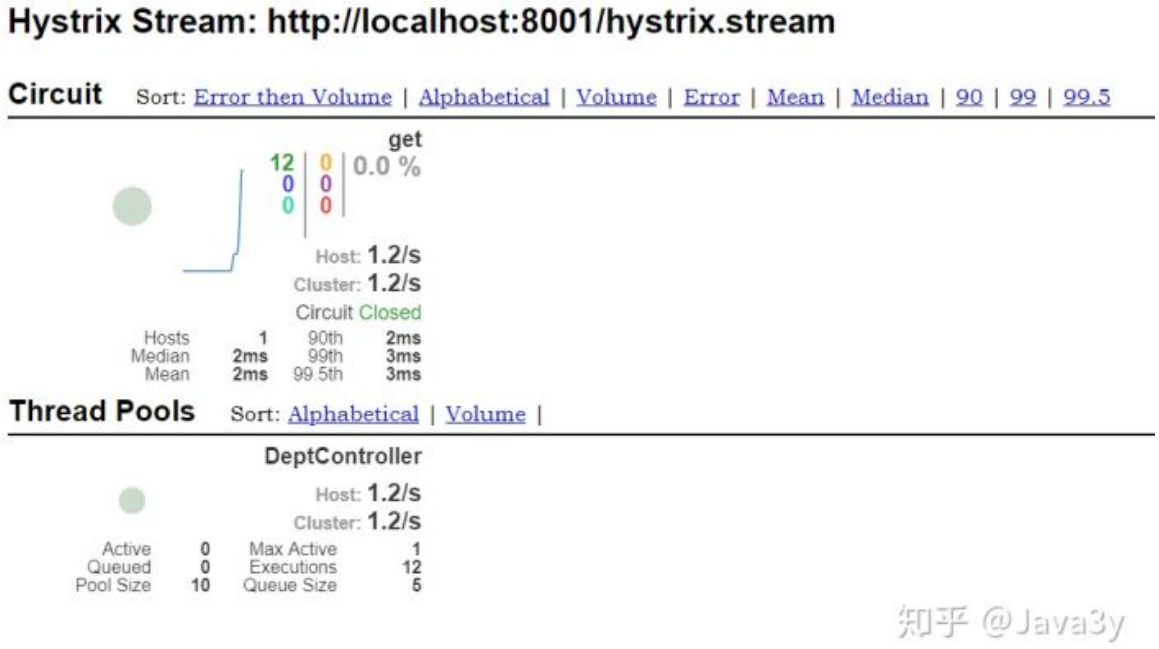
启动时的页面：


Hystrix Dashboard
http://hostname:port/turbine/turbine.stream
Cluster via Turbine (default cluster): http://turbine-hostname:port/turbine.stream
Cluster via Turbine (custom cluster): http://turbine-hostname:port/turbine.stream?cluster=[clusterName]
Single Hystrix App: http://hystrix-app:port/hystrix.stream
Delay: 2000 ms
Title: Example Hystrix App
Monitor Stream
知乎 @Java3y
Hystrix Dashboard
http://hostname:port/turbine/turbine.stream
Cluster via Turbine (default cluster): http://turbine-hostname:port/turbine.stream
Cluster via Turbine (custom cluster): http://turbine-hostname:port/turbine.stream?cluster=[clusterName]
Single Hystrix App: http://hystrix-app:port/hystrix.stream
Delay: 2000 ms
Title: Example Hystrix App
Monitor Stream
知乎 @Java3y

**监控单服务**的页面：

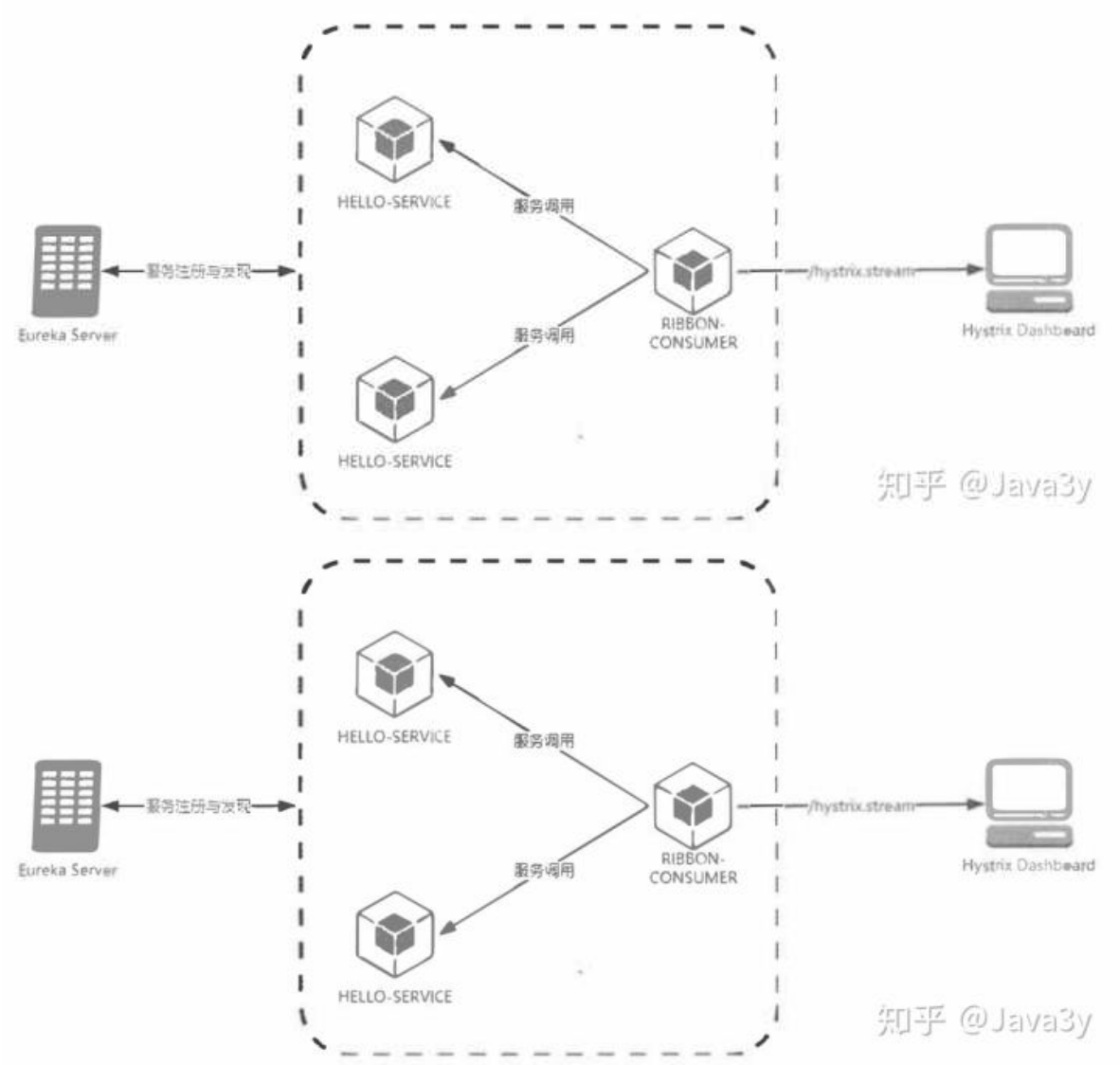




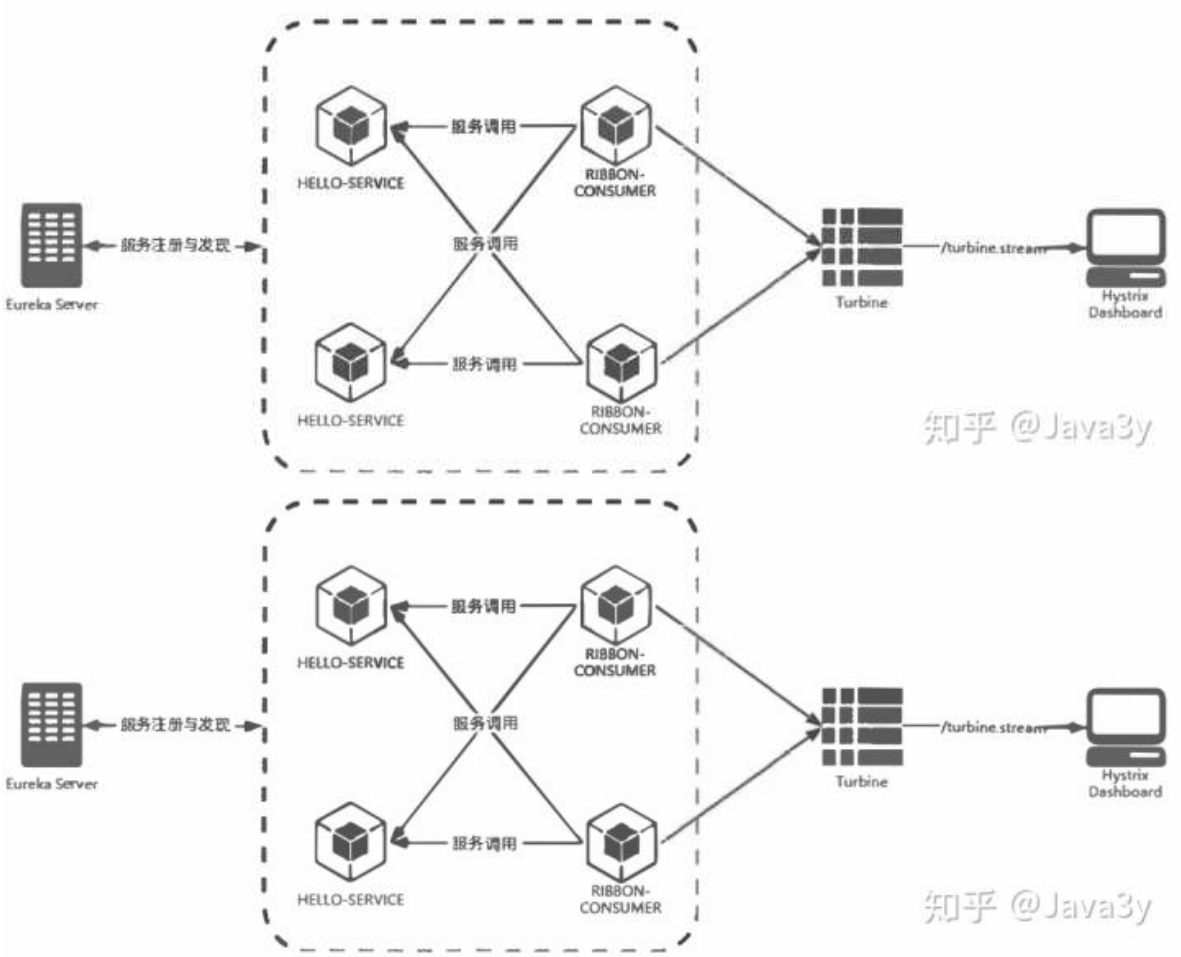






我们现在的服务是这样的：




除了可以开启单个实例的监控页面之外，还有一个监控端点 `/turbine.stream` 是对**集群**使用的。 从端点的命名中，可以引入Turbine, 通过它来**汇集监控信息**，并将聚合后的信息提供给 HystrixDashboard 来**集中展示和监控。**




举个例子：

- 3y和女朋友决定去万达玩，去到万达的停车场发现在负一层已经大大写上“负一层已停满，请下负二层，负二层空余停车位还有100个！”
- 这时，3y就跟女朋友说：“万达停车场是做得挺好的，如果它没有**直接**告知我负一层已满，可能我就去负一层找位置了，要是一堆人跑去负一层但都找不到车位的话，恐怕就塞死了”。3y接着说：“看停车位的状态也做得不错，在停车位上头有一个**感应(监控)**，如果是红色就代表已被停了，如果是绿色就说明停车位是空的”。
- 3y女朋友不屑的说：“你话是真的多”

参考资料：

- Hystrix，为什么说它是每个系统不可或缺的开源框架？https://zhuanlan.zhihu.com/p/34304136
- 深入理解Hystrix之文档翻译：https://zhuanlan.zhihu.com/p/28523060
- 谈谈我对服务熔断、服务降级的理解：https://blog.csdn.net/guwei9111986/article/details/51649240
- Hystrix几篇文章《青芒》：https://segmentfault.com/u/yedge/articles

# 8 引出Feign

上面已经介绍了Ribbon和Hystrix了，可以发现的是：他俩作为基础工具类框架**广泛地应用**在各个微服务的实现中。我们会发现对这两个框架的**使用几乎是同时出现**的。

为了**简化**我们的开发，Spring Cloud Feign出现了！它基于 Netflix Feign 实现，**整合**了 Spring Cloud Ribbon 与 Spring Cloud Hystrix, 除了整合这两者的强大功能之外，它还提 供了**声明式的服务调用**(不再通过RestTemplate)。

> Feign是一种声明式、模板化的HTTP客户端。在Spring Cloud中使用Feign, 我们可以做到使用HTTP请求远程服务时能与调用本地方法一样的编码体验，开发者完全感知不到这是远程方法，更感知不到这是个HTTP请求。

下面就简单看看Feign是怎么优雅地实现远程调用的：

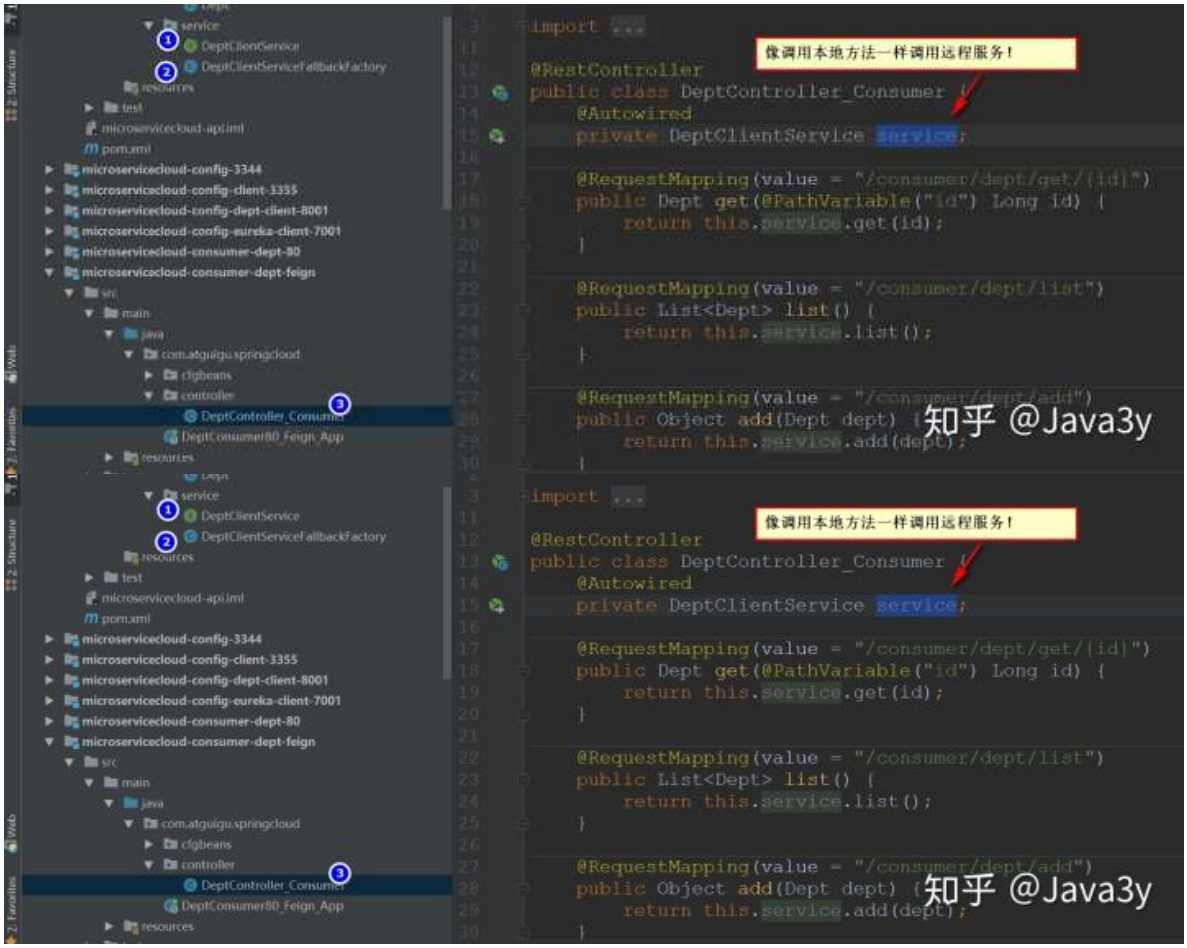
服务绑定：

```
// value --->指定调用哪个服务
// fallbackFactory--->熔断器的降级提示
@FeignClient(value = "MICROSERVICECLOUD-DEPT", fallbackFactory =
DeptClientServiceFallbackFactory.class)
public interface DeptClientService {


    // 采用Feign我们可以使用SpringMVC的注解来对服务进行绑定!
    @RequestMapping(value = "/dept/get/{id}", method = RequestMethod.GET)
    public Dept get(@PathVariable("id") long id);

    @RequestMapping(value = "/dept/list", method = RequestMethod.GET)
    public List<Dept> list();

    @RequestMapping(value = "/dept/add", method = RequestMethod.POST)
    public boolean add(Dept dept);
}
```

Feign中使用熔断器：

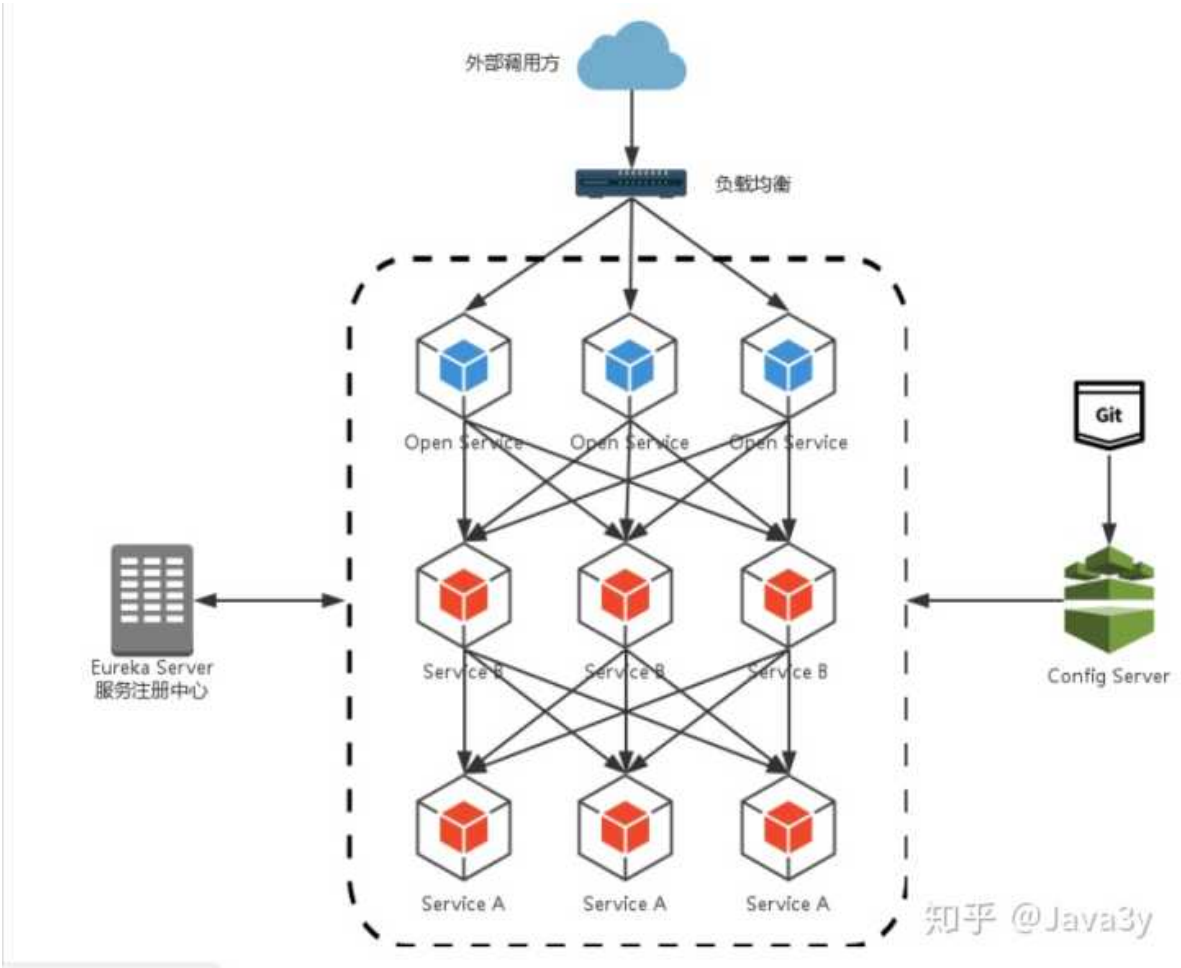
```
/**
 * Feign中使用断路器
 * 这里主要是处理异常出错的情况(降级/熔断时服务不可用，fallback就会找到这里来)
 */
@Component // 不要忘记添加，不要忘记添加
public class DeptClientServiceFallbackFactory implements
FallbackFactory<DeptClientService> {
    @Override
    public DeptClientService create(Throwable throwable) {
        return new DeptClientService() {
            @Override
            public Dept get(long id) {
                return new Dept().setDeptno(id).setDname("该ID：" + id + "没有没有
对应的信息,Consumer客户端提供的降级信息,此刻服务Provider已经关闭")
                        .setDb_source("no this database in MySQL");
            }

            @Override
            public List<Dept> list() {
                return null;
            }

            @Override
            public boolean add(Dept dept) {
                return false;
            }
        };
    }
}
```

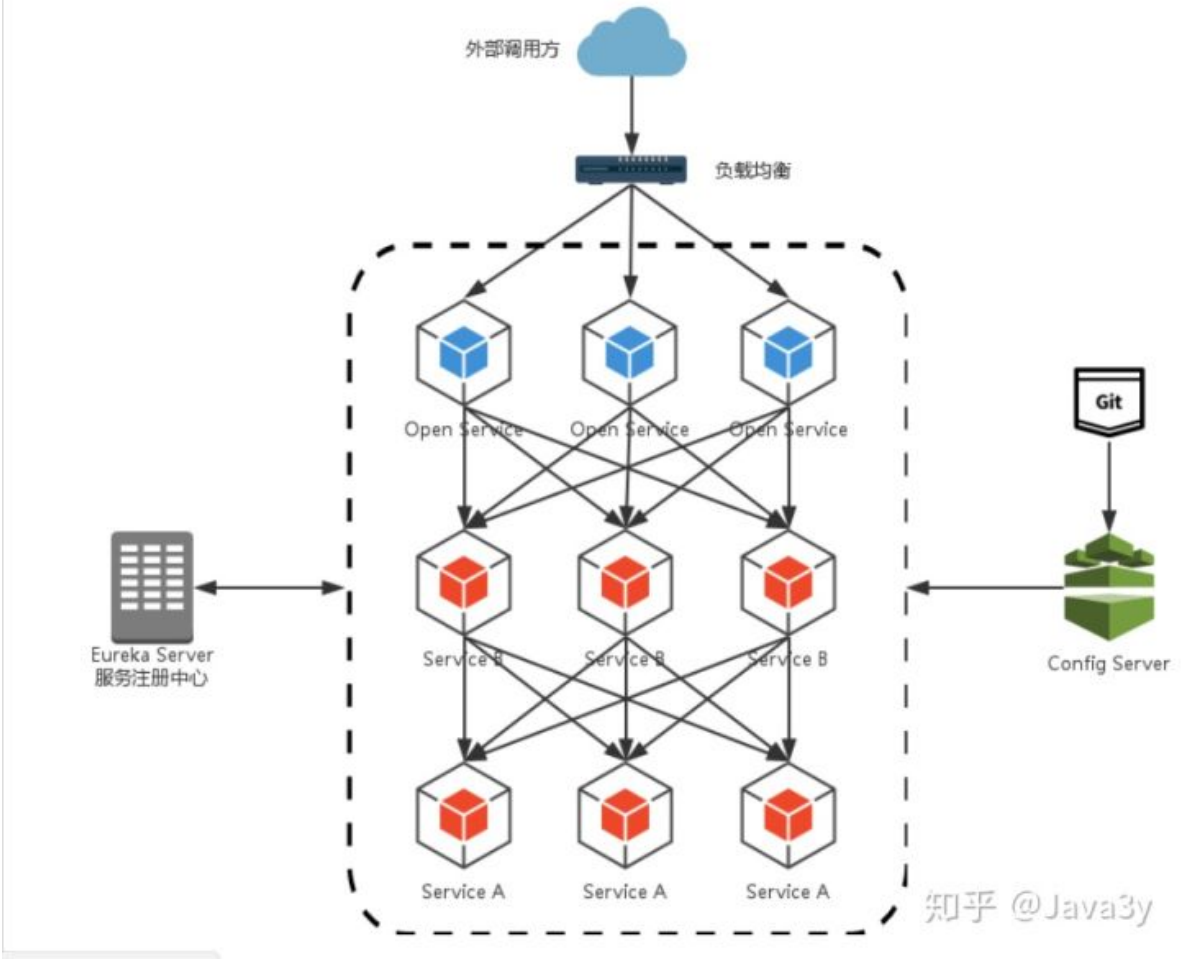
调用：




# 9 引出Zuul

基于上面的学习，我们现在的架构很可能会设计成这样：

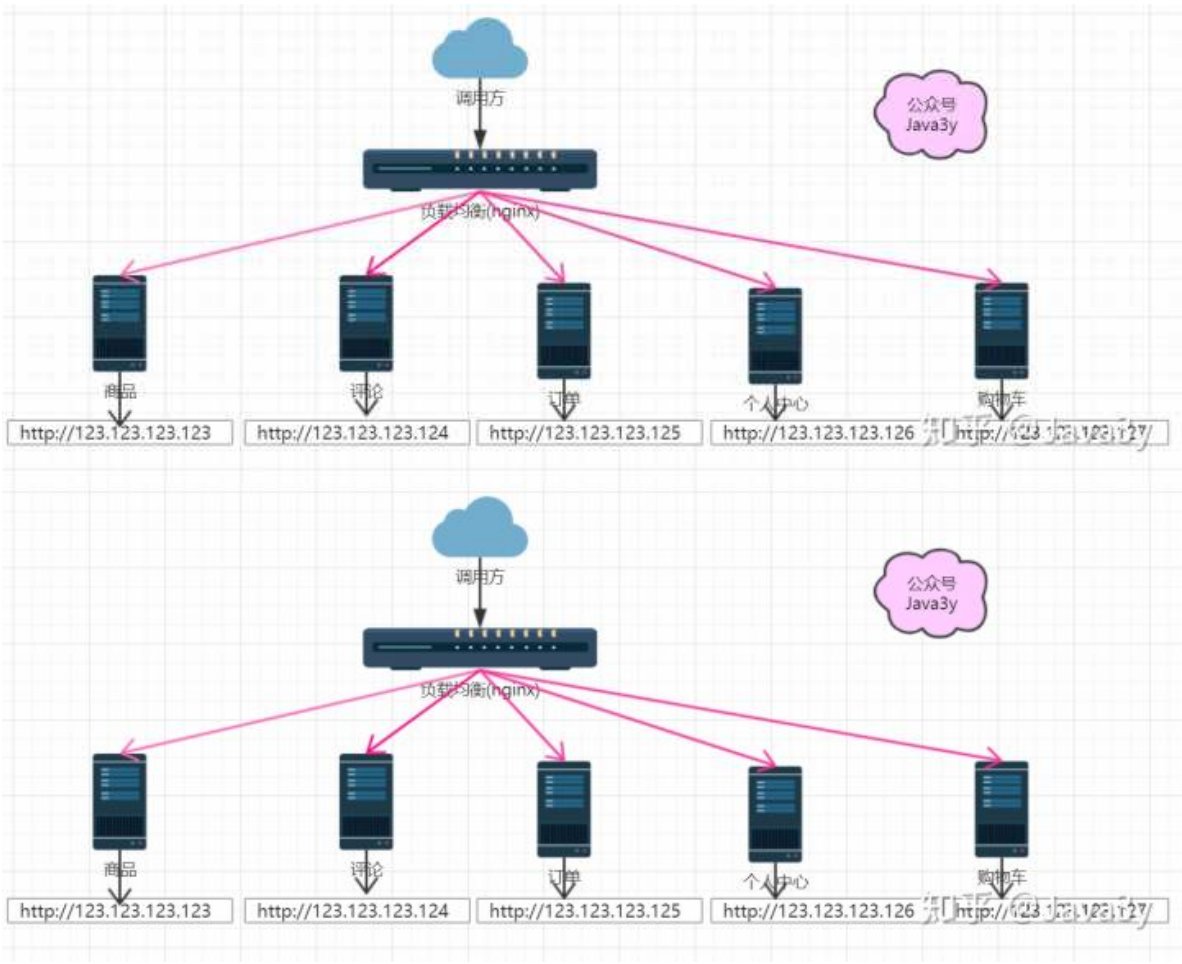



这样的架构会有两个比较麻烦的问题：

1. **路由规则与服务实例的维护问题**：外层的负载均衡(nginx)需要**维护**所有的服务实例清单(图上的OpenService)
2. **签名校验、 登录校验冗余问题**：为了保证对外服务的安全性， 我们在服务端实现的微服务接口，往往都会有一定的**权限校验机制**，但我们的服务是独立的，我们**不得不在这些应用中都实现这样一套校验逻辑**，这就会造成校验逻辑的冗余。

还是画个图来理解一下吧：




每个服务都有自己的IP地址，Nginx想要正确请求转发到服务上，就必须**维护着每个服务实例的地址**！

- 更是灾难的是：这些服务实例的IP地址还有可能会变，服务之间的划分也很可能会变。

```
http://123.123.123.123

http://123.123.123.124

http://123.123.123.125

http://123.123.123.126

http://123.123.123.127
```

购物车和订单模块都需要用户登录了才可以正常访问，基于现在的架构，只能在**购物车和订单模块都编写校验逻辑**，这无疑是冗余的代码。

为了解决上面这些常见的架构问题，**API网关**的概念应运而生。在SpringCloud中了提供了基于Netfl ix Zuul实现的API网关组件**Spring Cloud Zuul**。

Spring Cloud Zuul是这样解决上述两个问题的：

- SpringCloud Zuul通过与SpringCloud Eureka进行整合，将自身注册为Eureka服务治理下的应用，同时从Eureka中获得了所有其他微服务的实例信息。**外层调用都必须通过API网关**，使得**将维护服务实例的工作交给了服务治理框架自动完成。**
- 在API网关服务上进行统一调用来**对微服务接口做前置过滤**，以实现对微服务接口的**拦截和校验。**

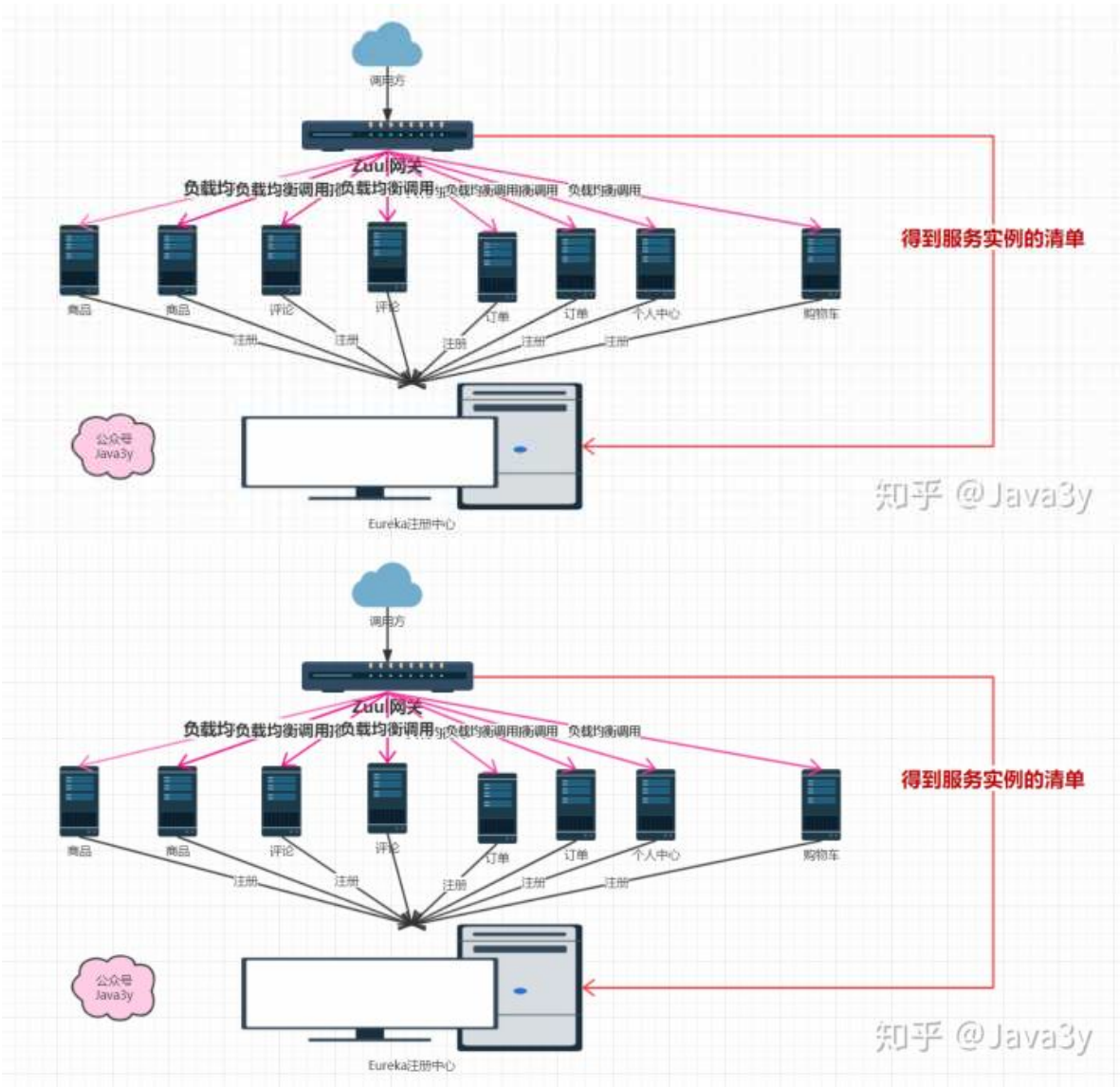
Zuul天生就拥有线程隔离和断路器的自我保护功能，以及对服务调用的客户端负载均衡功能。也就是说：**Zuul也是支持Hystrix和Ribbon。**

关于Zuul还有很多知识点(由于篇幅问题，这里我就不细说了)：

- 路由匹配(动态路由)
- 过滤器实现(动态过滤器)
- 默认会过滤掉Cookie与敏感的HTTP头信息(额外配置)

## 9.1可能对Zuul的疑问

Zuul支持Ribbon和Hystrix，也能够实现客户端的负载均衡。我们的Feign不也是实现客户端的负载均衡和Hystrix的吗？既然Zuul已经能够实现了，那我们的Feign还有必要吗？




或者可以这样理解：

- zuul是对外暴露的唯一接口相当于路由的是controller的请求，而Ribbonhe和Fegin路由了service的请求
- zuul做最外层请求的负载均衡，而Ribbon和Fegin做的是系统内部各个微服务的service的调用的负载均衡

有了Zuul，还需要Nginx吗？他俩可以一起使用吗？

- 我的理解：Zuul和Nginx是可以一起使用的(毕竟我们的Zuul也是可以搭成集群来实现高可用的)，要不要一起使用得看架构的复杂度了(业务)~~~

参考资料：

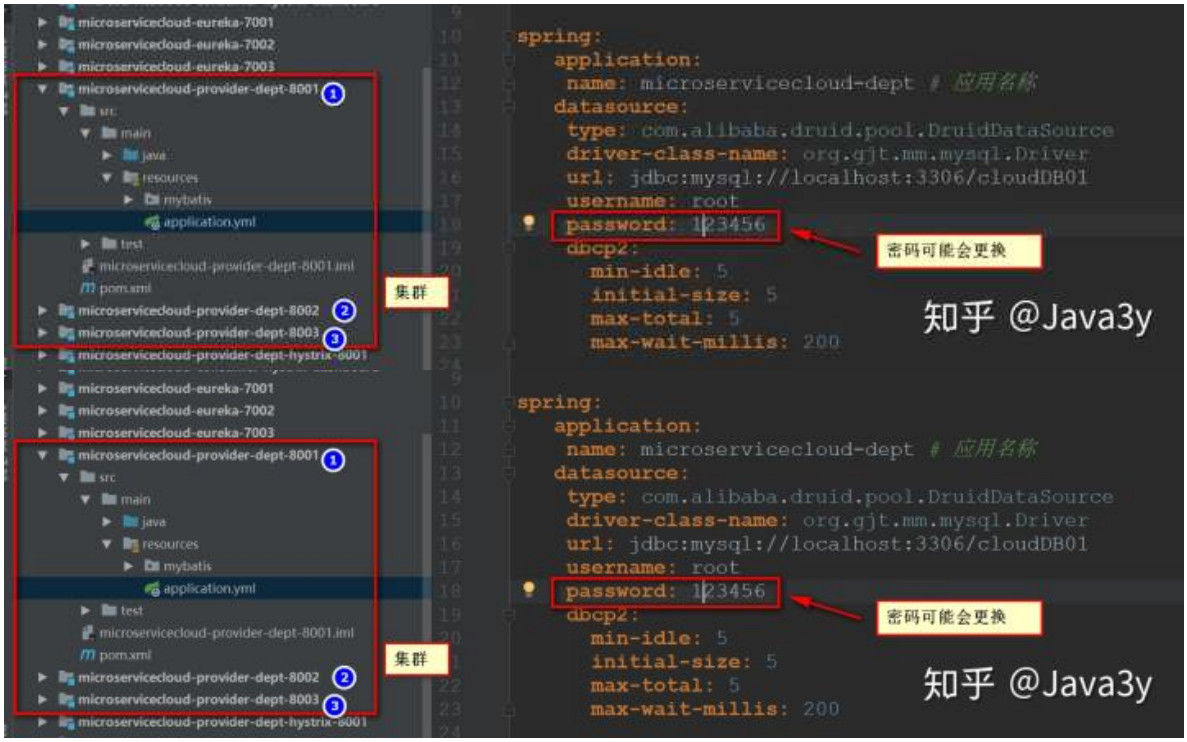
- 微服务与API网关（上）：为什么需要API网关？：http://blog.didispace.com/hzf-ms-apigateway-1/
- 谈谈 API 网关：https://www.jianshu.com/p/b52a2773e75f
- 谈谈微服务中的 API 网关（API Gateway）：https://www.cnblogs.com/savorboard/p/api-gateway.html
- API网关性能比较：NGINX vs. ZUUL vs. Spring Cloud Gateway：http://www.360doc.com/content/18/0208/05/46368139_728502763.shtml
- 谈API网关的背景、架构以及落地方案：http://www.infoq.com/cn/news/2016/07/API-background-architecture-floo
- zuul和nginx：https://zhuanlan.zhihu.com/p/37385481

# 10 引出SpringCloud Config

随着业务的扩展，我们的服务会越来越多，越来越多。每个服务都有自己的配置文件。

既然是配置文件，给我们配置的东西，那**难免会有些改动**的。

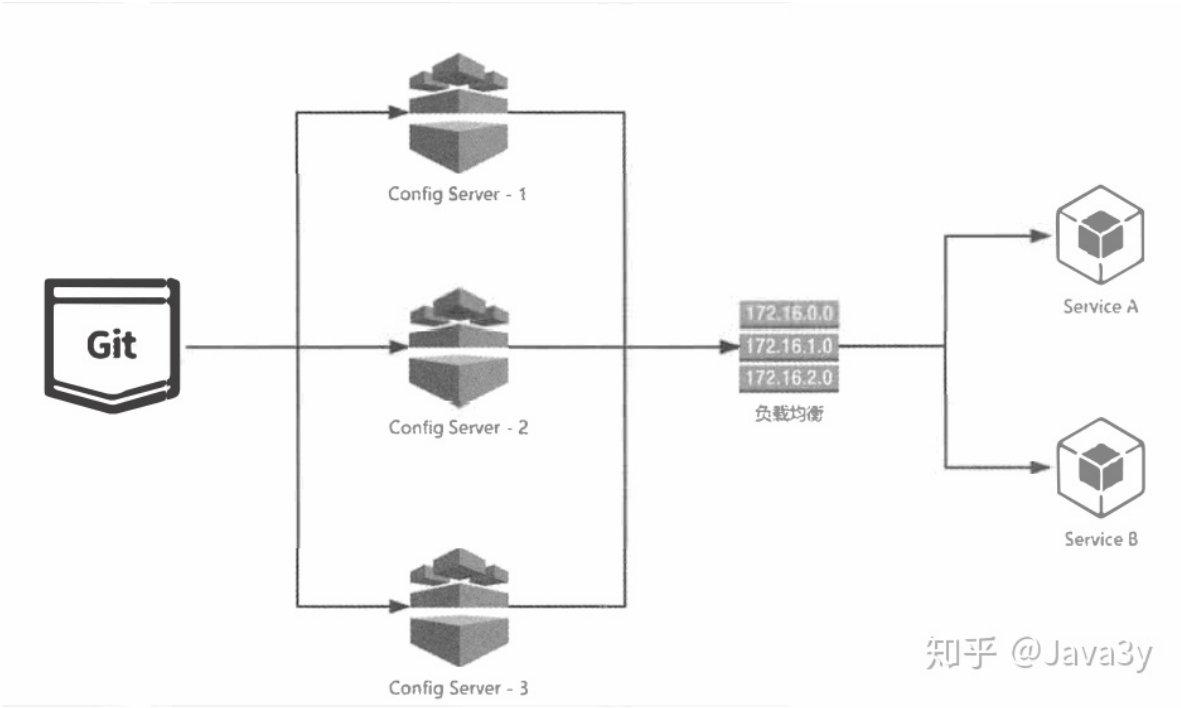
比如我们的Demo中，每个服务都写上**相同**的配置文件。万一我们有一天，配置文件中的密码需要更换了，那就得**三个都要重新更改**。

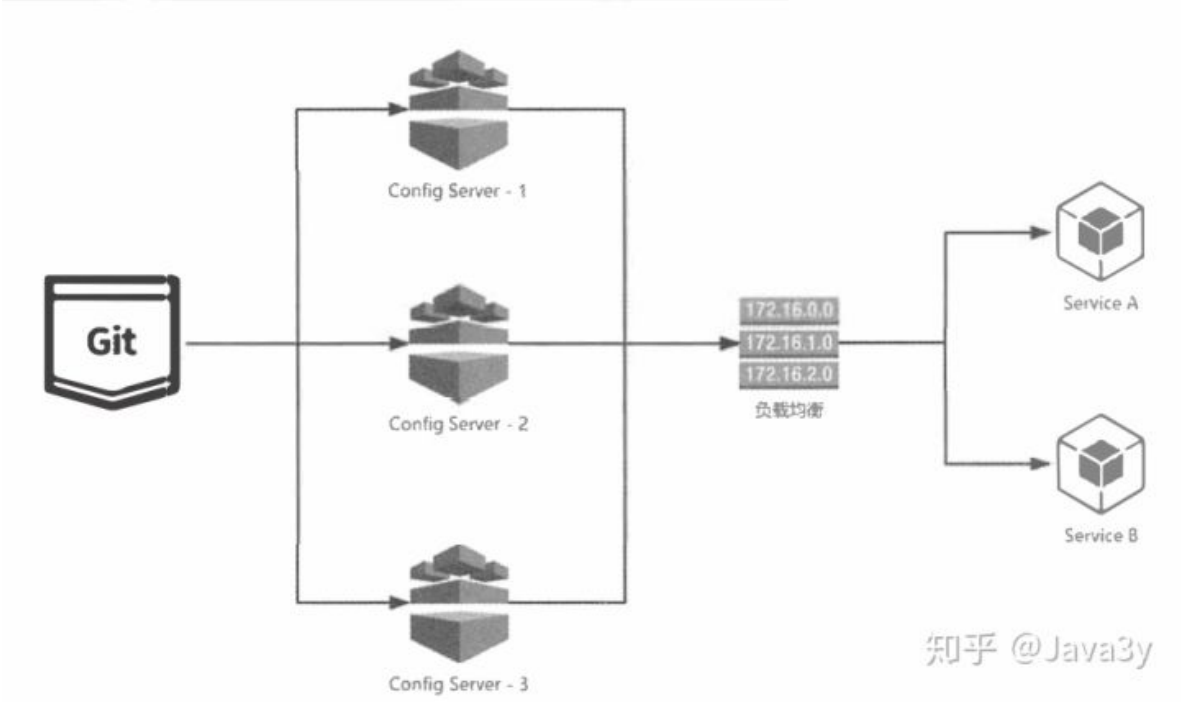



> 在分布式系统中，某一个基础服务信息变更，都**很可能**会引起一系列的更新和重启

Spring Cloud Config项目是一个解决分布式系统的配置管理方案。它包含了Client和Server两个部分，**server提供配置文件的存储、以接口的形式将配置文件的内容提供出去，client通过接口获取数据、并依据此数据初始化自己的应用。**

- 简单来说，使用Spring Cloud Config就是将配置文件放到**统一的位置管理**(比如GitHub)，客户端通过接口去获取这些配置文件。
- 在GitHub上修改了某个配置文件，应用加载的就是修改后的配置文件。




SpringCloud Config其他的知识：

- 在SpringCloud Config的服务端， 对于配置仓库的默认**实现采用了Git**，我们也可以配置SVN。
- 配置文件内的信息**加密和解密**
- 修改了配置文件，希望不用重启来**动态刷新配置**，配合Spring Cloud Bus 使用~

使用SpringCloud Config可能的疑问：application.yml和 bootstrap.yml区别

- https://www.cnblogs.com/BlogNetSpace/p/8469033.html

# 总结

本文主要写了SpringCloud的基础知识，希望大家看完能有所帮助~

SpringCloud的资料也很多，我整理一些我认为比较好，想要深入的同学不妨看看下边的资源~~~

SpringCloud系列文章参考资料：

- 史上最简单的 SpringCloud 教程 | 终章https://blog.csdn.net/forezp/article/details/70148833
- Spring Cloud基础教程《程序员DD》http://blog.didispace.com/Spring-Cloud%E5%9F%BA%E7%A1%80%E6%95%99%E7%A8%8B/
- Spring Cloud 系列文章《纯洁的微笑》：http://www.ityouknow.com/spring-cloud.html
- SpringCloud系列文章：https://www.cnblogs.com/woshimrf/tag/SpringCloud/
- SpringCloud系列文章《狂小白》：https://www.cnblogs.com/huangjuncong/tag/SpringCloud/
- SpringCloud官方文档：http://projects.spring.io/spring-cloud/
- Spring Cloud 中文文档：https://springcloud.cc/spring-cloud-dalston.html#_appendix_compendium_of_configuration_properties

参考书籍：

- 《SpringCloud 微服务实战》

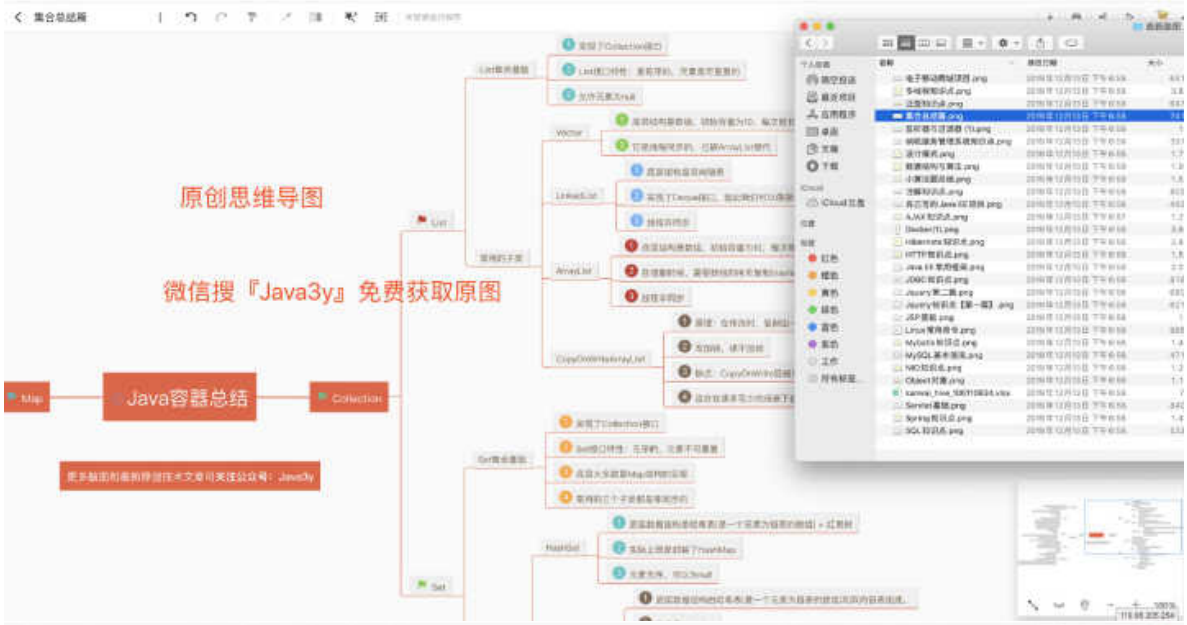
SpringCloud GitHub Demo(看完文章的同学可以自己练手玩玩，写好了ReadMe了)：

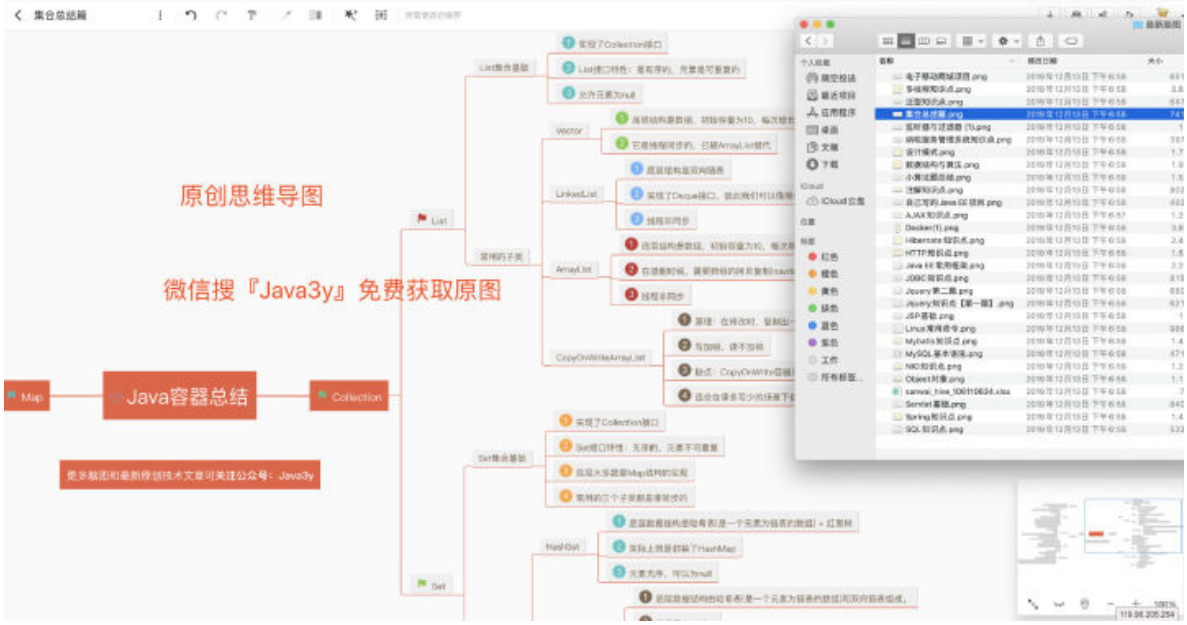
- https://github.com/ZhongFuCheng3y/msc-Demo

涵盖Java后端所有知识点的开源项目（已有5.8K star）：https://github.com/ZhongFuCheng3y/3y

如果大家想要**实时**关注我更新的文章以及分享的干货的话，微信搜索**Java3y**

PDF文档的内容**均为手打**，有任何的不懂都可以直接**来问我**（公众号有我的联系方式）。


集合总结篇
原创思维导图
微信搜『Java3y』免费获取原图
Java容器总结
Map
Collection
List
Set
更多脑图和最新原创技术文章可关注公众号：Java3y


集合总结篇
原创思维导图
微信搜『Java3y』免费获取原图
Java容器总结
Map
Collection
List
Set
更多脑图和最新原创技术文章可关注公众号：Java3y


纯手打
公众号回复「888」免费获取
各种案例和图
加油！